Funções nativas do Oracle aplicadas no Delphi: Busca fonética e por similaridade

Clube Delphi

Cobrança Bancária com ACBr Boleto
Entenda como funciona e aplique já

Mapeamento de classes com Aurelius
Cenários de aplicação

Ano 13 :: Edição 151 :: R$ 14,90 DevMedia

EXPLORANDO O DataSnap

Um teste de performance

Testes unitários automatizados
DUnit na prática para apoiar agilidade

Introdução à Biblioteca FireDAC
Conheça a nova biblioteca de componentes

ISSN 1517990-7

VOCÊ É ASSINANTE MVP?

ENTÃO PODE TER CERTEZA: O MERCADO PREFERE VOCÊ!

DESENVOLVEDOR ATUALIZADO É O MAIS VALIOSO DO MERCADO!

Muitos profissionais estão acomodados, não se atualizam, não leem livros e não vão a eventos. Acham que é a empresa que deve mantê-los atualizados. Gastam dinheiro com tudo, mas acham caro investir menos de R$60,00 por mês em suas carreiras.

Mas você não! Você é MVP! Parabéns!

TENHA ACESSO A:

- +DE 260 CURSOS ONLINE
- 09 REVISTAS MENSAIS
- 7.850 VÍDEO-AULAS

POR APENAS 59,90 MENSAIS

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

QUEM TEM ESTÁ TRANQUILO.

DevMedia

Acesse: www.devmedia.com.br/mvp

Sumário

06 – Colocando um Servidor DataSnap à Prova
[*Roberto Schneiders*]

Conteúdo sobre Boas práticas

13 – Testes Unitários Automatizados
[*Diego Garcia*]

Conteúdo sobre Novidades

22 – Introdução à Biblioteca FireDAC
[*Fabrício Hissao Kawata*]

Artigo no estilo Mentoring

30 – Emissão de Boletos Bancários com ACBr
[*Giuliano Scombatti*]

Artigo no estilo Solução Completa

36 – Use Funções Nativas do Oracle Aplicadas no Delphi
[*Matheus de Lima Lyra*]

Conteúdo sobre Boas Práticas

44 – Mapeamento ORM com TMS Aurelius
[*Murilo Aparecido Panosso*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!

Faça já sua assinatura digital ! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Clube Delphi

Ano 13 - 151ª Edição 2013 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral

Paulo Quicoli
pauloquicoli@gmail.com

Equipe Editorial

Guinther Pauli
guintherpauli@gmail.com

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br

Giuliano Scombatti Pinto
giuliano@sygnux.com.br

Daniel Sobrinho Laporte
daniel.laporte@gmail.com

Jornalista Responsável

Kaline Dolabella - JP24185

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

EDITORIAL

A Embarcadero tem abordado diferentes mercados nos últimos anos. Seu foco recentemente está voltado para o desenvolvimento Mobile. Contudo, um framework que sempre está presente é o DataSnap. Sempre é apresentado como a solução para todos os problemas. Nosso colega Roberto Scheneiders apresenta em seu artigo uma pesquisa comparativa entre o DataSnap, um framework Java, o WCF do .NET Framework e um framework independente desenvolvimento em Delphi chamado mORMot. Todos são colocados à prova como servidores de aplicação, simulando um acesso realmente grande de acessos, simultâneos ou não. O resultado para muitos será uma surpresa, leia o artigo.

O Fabrício continua apresentando para nós as novidades criadas, desta vez temos um overview completo do FireDAC. Veja como utilizá-lo para situações comuns como conexão com Firebird, ordenação de dados em memória, filtros, campos calculados, mestre-detalhe e muito mais. Após esse artigo você estará pronto para começar a utilizar o FireDAC em seus próprios projetos.

Voltando agora os olhos para situações reais de vários desenvolvedores, o Giuliano preparou o que posso chamar "must read" para aqueles que precisam integrar geração e administração de boletos em seus sistemas comerciais. Um braço da suíte ACBr é apresentado, o ACBr Boleto. Além de ser apresentada as funcionalidades, algumas características desse tipo de cobrança são expostas, tirando assim várias dúvidas dos menos experientes no assunto. Não perca.

Outro assunto que tem chamado a atenção dos desenvolvedores Delphi é a questão do ORM, principalmente no uso do framework Aurelius. Quando se fala de ORM é preciso entender muito bem sobre como mapear uma classe, ou um conjunto de classes, para um banco de dados, pois, é nisso que um framework ORM se baseia para gerar o acesso ao banco de dados. Assim, um mapeamento incorreto trará problemas para a aplicação, seja informações incorretas ou até mesmo perda de desempenho. O artigo trazido pelo Murilo faz uma abordagem de vários cenários de mapeamento, clarificando o assunto para quem deseja caminhar essa estrada.

E por fim, temos um artigo de banco de dados, apresentado pelo Matheus Lima. Veja como utilizar recursos específicos do Oracle para criar consultas textuais inteligentes. Entenda o mecanismo de sugestão de textos, similar ao que o Google aplica em consultas, e como utilizar tudo isso com Delphi, em um exemplo simples mas funcional.

Boa leitura!

Paulo Quicoli
pauloquicoli@gmail.com
Editor Geral
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

DevMedia

Seu **conhecimento vale muito** na DevMedia

Publique artigos e vídeos gratuitamente no site e participe do programa **PayContent.**

Sua participação pode ser premiada: quanto mais você postar, mais chances de ganhar.

Para saber mais acesse:
www.devmedia.com.br/paycontent

DevMedia

Colocando um Servidor DataSnap à Prova

Uma análise de performance e estabilidade

O modelo cliente-servidor ainda é uma realidade para muitos softwares atuais. Este modelo atende bem a necessidade de softwares desktop, mas apresenta problemas a partir do momento em que se tem a necessidade de oferecer outras interfaces, como Web e Mobile. Ter diversas interfaces se conectando diretamente a uma base de dados traz grandes desafios que geralmente não compensam os riscos em softwares de grande porte. O mais indicado na maioria dos casos é mudar o software para um modelo multicamadas.

O modelo multicamadas nos permite separar diferentes partes do software tornando mais simples oferecer diversas interfaces para um único domínio de negócio. Um exemplo de um modelo multicamadas simples, pode ser representado por três camadas: base de dados, regras de negócio e interface. A camada onde se concentram as regras de negócio é uma camada servidora, que vai fornecer dados e serviços às camadas mais externas (interfaces).

Uma mudança no modelo da arquitetura pode oferecer muitos desafios. A adição de camadas em uma arquitetura geralmente acrescenta complexidade, principalmente, na camada de comunicação entre as camadas. Desenvolver um software servidor também possui particularidades que uma empresa acostumada a trabalhar somente com softwares desktop pode não estar preparada para enfrentar. Gerenciamento de memória e desenvolvimento multithread passam a ser cruciais. Vazamentos de memória podem não ser um grande problema em um software desktop, mas certamente será em um servidor. A análise detalhada neste artigo vai avaliar o gerenciamento de memória e threads de algumas APIs.

Para softwares Delphi, a tecnologia mais difundida para modelos multicamadas é o DataSnap, desenvolvido pela Embarcadero. As últimas versões do Delphi trouxeram grandes mudanças e melhorias.

Objetivos da análise

O objetivo dessa análise foi levar a tecnologia DataSnap ao seu limite. Descobrir seus limites nos permitiu avaliar com mais precisão quais requisitos

Resumo DevMan

Colocando um Servidor DataSnap à Prova:

Neste artigo apresentaremos uma análise de performance e estabilidade da tecnologia DataSnap. As últimas versões do Delphi trouxeram mudanças e melhorias para a tecnologia que é a mais popular em softwares multicamadas para Delphi. A análise vai mostrar até que ponto a tecnologia é capaz de suportar aplicações com grande volume de requisições usando REST. Em eventos da comunidade Delphi, geralmente organizados pela Embarcadero, como a Delphi Conference e Webinars, a tecnologia DataSnap aparece como unanimidade entre os especialistas. É comum assistirmos palestras de profissionais que contam experiências fantásticas com essa tecnologia e recomendam-na para qualquer software Delphi. Contudo temos aqui uma análise que visa esclarecer até que ponto a tecnologia é realmente uma escolha óbvia para softwares Delphi e até onde a tecnologia atende as necessidades de um software grande. Essa análise foi motivada como estudo de viabilidade para um projeto de software. É importante fazer testes antes de começar a usar uma API. Não se pode escolher uma API que será responsável por toda comunicação entre as camadas do seu software sem ter total consciência do que ela é capaz de suportar. Há muito a ser avaliado na escolha de um framework desse tipo. É necessário assegurar-se que ele atende os requisitos de desempenho, estabilidade, escalabilidade e recursos que serão necessários no projeto. Mudar um framework pode ser extremamente complicado no meio de um projeto e muitas vezes é simplesmente inviável. É muito mais barato testar e conhecer o que você vai usar antes de colocar ele em seu projeto. Às vezes uma tecnologia parece excelente em uma primeira análise e mais tarde você percebe que não serve para o projeto.

Antes de adotar um framework, API, biblioteca, componente, faça testes de estresse e provas de conceito. Submeta a tecnologia ao máximo de estresse que for possível, encontre o limite dessa tecnologia, só assim é possível ter condição de avaliar se a tecnologia serve ou não para o seu projeto. É exatamente isso que fizemos com o DataSnap.

Em que situação o tema é útil:

Quando se está avaliando a migração de um software Delphi que usa a estrutura cliente-servidor para multicamadas ou está preocupado com a escalabilidade e performance de seu software multicamadas.

a tecnologia é capaz de atender e quais os ambientes onde a tecnologia é recomendada.

Os testes realizados buscaram medir os índices de performance e estabilidade em ambientes críticos, com um alto número de requisições ao servidor.

Metodologia

Os testes se basearam em um servidor fornecendo um método simples via REST. O método disponibilizado pelo servidor não realiza qualquer processamento ou alocação de memória, simplesmente retorna o texto "Hello World". Dessa forma a API será estressada ao máximo, não havendo influência de qualquer outra implementação.

Além do servidor DataSnap, preparamos servidores com outras tecnologias. As tecnologias não necessariamente possuem o mesmo propósito, mas todas oferecem uma API REST. São elas (veja seção **Links**):

- mORMot (Delphi)
- ASP.NET WCF
- Jersey/Grizzly (Java)
- Node.Js (JavaScript)

São tecnologias para diferentes plataformas e linguagens. Elas vão nos permitir ter uma base de comparação com o DataSnap. Essas tecnologias e linguagens não foram estudadas a fundo e estes softwares foram elaborados com um conhecimento mínimo.

O ambiente e hardware utilizados são compostos por dois computadores intermediários conectados através de uma rede gigabit. A **Figura 1** mostra com mais detalhes esse ambiente.

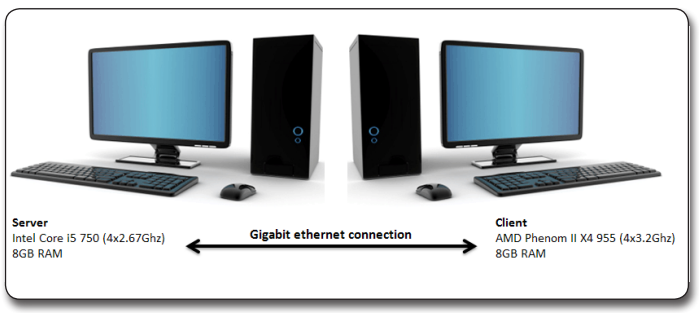


Figura 1. Ambiente de testes

No lado cliente utilizou-se o software JMeter para efetuar as requisições ao servidor. O JMeter é uma ferramenta especializada em testes para servidores, desenvolvida em java. Essa ferramenta propicia uma série de informações relevantes, que foram utilizadas na análise.

A informação que serviu como base para a avaliação de desempenho foi a de requisições por segundo. O consumo de memória foi obtido através do software ProcessExplorer (ver seção **Links**) no final de cada execução.

Os testes realizados são de dois tipos, com e sem concorrência nas requisições. No teste sem concorrência, apenas um cliente realizava as requisições para o servidor. No teste com concorrência a aplicação cliente possui 50 e 100 threads enviando requisições ao mesmo tempo.

Versões

A análise apresenta resultados de quatro diferentes servidores usando a tecnologia DataSnap, são eles:

- DataSnap (Console) XE3: Aplicação console compilada no Delphi XE3;
- DataSnap (Console) XE3u1: Aplicação console compilada com o Delphi XE3 Update 1;
- DataSnap (ISAPI) XE3u1: Aplicação ISAPI compilada com o Delphi XE3 Update 1;
- DataSnap (VCL) XE3u1 w/ optimizations: Aplicação VCL compilada com o Delphi XE3 Update 1 com as otimizações.

Os testes mostraram problemas de estabilidade no DataSnap quando submetido a um teste de carga com concorrência. A partir desses testes a Embarcadero liberou correções no Update 1 da versão XE3 para corrigir os problemas. Correções estas que não foram disponibilizadas para versões anteriores.

Um dos servidores DataSnap foi compilado no Delphi XE3 (sem update) para efeito de comparação com o Delphi XE3 Update 1.

Nota

Não há dados para os testes de carga com concorrência para o servidor DataSnap (Console) compilado com o Delphi XE3 porque ele é incapaz de rodá-los devido aos problemas de estabilidade.

Otimizações para o servidor DataSnap

Existe uma série de otimizações que podem ser aplicadas a um servidor DataSnap. Geralmente as configurações padrões dos componentes e códigos gerados pelos wizards do Delphi possuem configurações suficientemente boas para funcionar, mas às vezes pouco otimizadas. É importante buscar informações sobre configurações dos componentes que se está usando para buscar um maior conhecimento da API. Saber como ela funciona ajuda na hora de fazer algum ajuste fino.

Uma boa forma de começar a conhecer a API do DataSnap é montar um servidor sem a ajuda do wizard do Delphi. Será necessário conhecer qual a responsabilidade de cada componente e como ele se comunica com os demais. Conhecer cada componente da API e como eles se inter-relacionam vai fazer diferença na hora de buscar otimizações. Não veremos todos os tipos de otimizações para o DataSnap, são apresentados alguns, que possuem influência direta em estabilidade e desempenho do servidor.

HTTP Keep-Alive

Em uma comunicação HTTP, cada requisição enviada ao servidor requer a abertura de uma conexão. Assim que a requisição for respondida, essa conexão é fechada. Desse modo, para cada requisição cria-se uma nova conexão. O procedimento de abertura e fechamento de uma conexão HTTP pode afetar o desempenho

das aplicações em casos onde a frequência de requisições é muito alta, como nos testes apresentados nesse artigo.

O Keep-Alive é um recurso do protocolo HTTP que tem como objetivo diminuir a quantidade de novas conexões que são estabelecidas com o servidor. Quando a aplicação cliente e servidor possuem o recurso Keep-Alive uma conexão pode receber diversas requisições, poupando tempo e processamento.

Por outro lado, uma requisição ao servidor deixará sempre uma conexão aberta por um determinado tempo. Se o software cliente não estiver fazendo requisições em uma frequência alta, isso pode afetar negativamente o desempenho do servidor. É importante se aprofundar no assunto e sempre efetuar testes de estresse de acordo com situações reais de cada projeto.

Ao contrário das demais tecnologias, essa opção vem desativada por padrão no DataSnap. Para ativá-la basta alterar a propriedade Keep-Alive do IdHTTPWebBrokerBridge, que é um componente da API.

Infelizmente, essa otimização não foi aplicada nos servidores DataSnap usados nos testes. Inexplicavelmente houve uma queda muito grande na velocidade do servidor. Todos os demais servidores estão usando Keep-Alive e obtiveram ganhos de desempenho significativos. Aparentemente é algum bug na API que deve ser corrigido pela Embarcadero no futuro.

Consumo de memória

Consumo de memória é sempre importante, mas em servidores é crítico. Servidores devem funcionar 24 horas por dia, 7 dias por semana. Devem funcionar anos a fio sem a necessidade de ser reiniciado.

Se tratando de Delphi, onde o ciclo de vida dos objetos é de responsabilidade do programador, é comum haver problemas de vazamento de memória (memory leak), quando objetos não são destruídos após sua utilização. Em aplicações desktop isso pode não ser um grande problema, por que frequentemente o sistema é reiniciado. Em um servidor, um único vazamento de memória em qualquer método pode fazer sua aplicação entrar em colapso depois de alguns dias funcionando.

Para quem está migrando de um modelo cliente/servidor para multicamadas, talvez, terá que se reeducar com relação a isso. Desenvolver para servidor requer um cuidado especial com a quantidade de memória que é alocada. Todo e qualquer novo método inserido no servidor deve ser avaliado. Um método consumindo muita memória pode ser requisitado por centenas de clientes ao mesmo tempo e fazer com que o servidor use toda memória disponível, o que fará o servidor entrar em colapso. Em aplicações 32 bits isso é ainda mais crítico, porque você terá no máximo 4Gb de memória para utilizar. Isso leva a um problema de escalabilidade da aplicação, que pode ser ainda mais difícil de resolver.

Se todo cuidado é pouco com relação a consumo de memória no servidor, também é um item importante da API. Se a API tiver um gerenciamento instável de memória, isso vai afetar sua camada de negócio. Um servidor parado significa a camada de negócio parada, o que é semelhante a uma falha no banco de dados em um modelo cliente/servidor. Se isso acontecer, você estará encrencado.

A tecnologia REST propõem serviços que não mantém estado (Stateless), ou seja, o servidor não deve guardar dados da aplicação cliente em sessões. Na prática isso é mais complicado, porque às vezes se precisa dessas informações. De qualquer maneira, internamente o DataSnap possui sessões, que parecem não ter utilidade para o servidor em si mas devem ser importantes para a API, já que não se pode desativá-las.

Uma sessão é criada a cada requisição REST ao servidor. Curiosamente, essa sessão não é reaproveitada, mesmo quando o mesmo cliente faz outra requisição. Essa sessão possui um timeout em torno de 20 minutos e não é destruída automaticamente. Dessa forma, requisições ao servidor significam novas sessões, que só são destruídas quando alcançarem o tempo determinado pelo timeout, ou seja, 20 minutos. Todas as requisições efetuadas em 20 minutos irão criar sessões e nenhuma será destruída.

Obviamente isso causará um consumo excessivo e desnecessário de memória. Nos testes notou-se um aumento constante no uso de memória do servidor nos primeiros 20 minutos de execução. Após os 20 minutos iniciais as sessões começam a ser destruídas devido ao timeout. Além do consumo de memória existe o problema de overhead que essas sessões trazem. A criação e destruição delas afetam negativamente o desempenho do servidor, mas não se encontrou uma maneira de evitar isso.

Não há como desativar essas sessões e evitar o overhead, mas existe uma forma de forçar a destruição dela ao final da execução de um método, diminuindo a quantidade de memória consumida pelo servidor. A cada requisição feita ao servidor, uma sessão será criada e destruída. Daniele Teti escreveu um artigo no seu blog (ver seção **Links**) explicando essa técnica. A **Listagem 1** mostra um exemplo de um método simples forçando a destruição de uma sessão.

Listagem 1. Forçando a destruição das sessões internas do DataSnap

```
uses System.StrUtils, DataSnap.DSSession, Data.DBXPlatform;

function TServerMethods1.HelloWorld: String;
begin
  Result := 'Hello World';
  GetInvocationMetaData.CloseSession := True;
end;
```

Thread Pool

Por padrão a tecnologia DataSnap cria uma thread para responder cada requisição HTTP. Isso pode causar um problema de overhead quando se tem muitas conexões. Pode-se mudar isso criando um Thread Pool, que basicamente é uma lista de threads criadas para responder as requisições. Essas threads são criadas na inicialização da aplicação e destruídas somente quando o aplicativo for fechado. Teoricamente haverá sempre uma lista de threads pronta para atender as requisições e não haverá criação

e destruição de threads para cada requisição. A **Listagem 2** mostra a implementação de um thread pool com 50 threads. Esse código foi retirado do blog do Marco Cantù (ver seção **Links**)

Listagem 2. Criação de um Thread Pool

```
var
  SchedulerOfThreadPool: TIdSchedulerOf
  ThreadPool;
begin
  FServer := TIdHTTPWebBrokerBridge.Create(Self);

  SchedulerOfThreadPool := TIdSchedulerOf
  ThreadPool.Create(FServer);
  SchedulerOfThreadPool.PoolSize := 50;
  FServer.Scheduler := SchedulerOfThreadPool;

  FServer.MaxConnections := 150;
end;
```

A propriedade MaxConnections do servidor também pode ser alterada conforme a necessidade. Marco Cantù sugere usar 150 no exemplo, mas isso depende do ambiente em que o servidor será utilizado.

Testes sem concorrência

Nesse teste, há uma aplicação cliente com somente uma thread, enviando requisições sequencialmente, uma após a outra. O servidor só recebe uma requisição por vez, sem haver qualquer concorrência.

O objetivo deste teste é verificar o comportamento do servidor em um ambiente menos crítico que também servirá de base para os outros testes. Foram efetuados testes com 100 mil e 1 milhão de requisições (**Figura 2**).

É possível ver o resultado do primeiro teste e esse revela uma diferença grande entre as demais tecnologias e o DataSnap. Todos os servidores DataSnap obtiveram resultados semelhantes, mas extremamente baixos, já que os demais servidores foram em torno de 12 vezes mais rápidos. Os resultados semelhantes entre as demais tecnologias mostra que todas atingiram o limite imposto pelo ambiente e hardware utilizados.

Como o desempenho foi idêntico para os testes com 100 mil e 1 milhão de requisições, os gráficos mostram somente os resultados dos segundo teste.

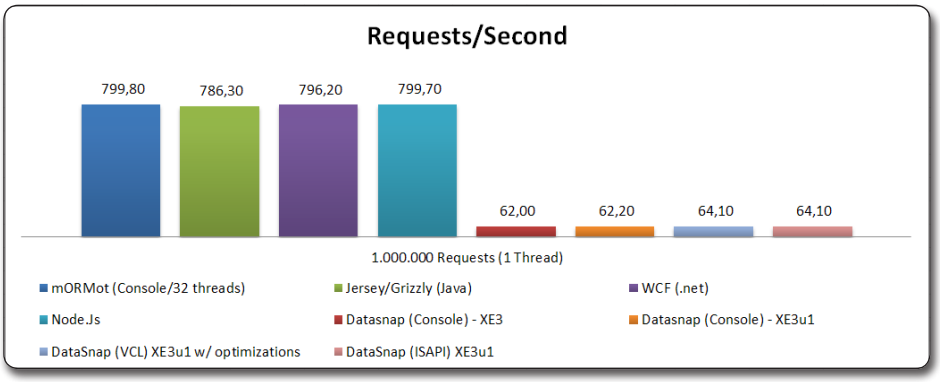


Figura 2. Teste de desempenho com 1 milhão de requisições, uma thread

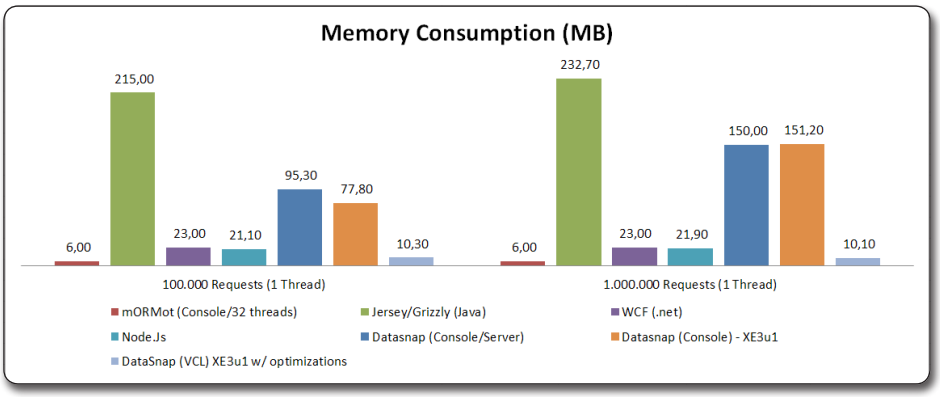


Figura 3. Consumo de memória nos testes com uma thread

Quanto ao consumo de memória, obtiveram-se resultados diferentes nos dois testes, a **Figura 3** exibe os dados de ambos os testes.

Os resultados indicam diferentes comportamentos entre os servidores DataSnap. O servidor que não possui as otimizações mostrou um consumo progressivo que aumenta conforme o número de requisições. Esse comportamento é perigoso porque pode fazer com que o servidor consuma toda memória disponível e entre em colapso. É importante salientar que o consumo avaliado indica somente o que a API está consumindo, já que o método implementado no servidor não realiza alocação de memória. Em uma aplicação real o consumo será naturalmente maior.

O servidor com otimizações mostrou um consumo baixo, apesar de ter variado durante o teste. Isso demonstra que a otimização realizada visando diminuir o consumo de memória fez o efeito esperado. Forçar a destruição das sessões parece reduzir consideravelmente o consumo de memória, mas é importante salientar que durante o teste o consumo varia, chegando a níveis mais elevados do que o apresentado nos gráficos.

Durante os testes verificou-se um comportamento, por vezes, imprevisível com relação ao consumo de memória dos servidores DataSnap. É possível notar que no primeiro teste o servidor compilado com o XE3 Update 1 consome menos memória (77,8MB) que o servidor compilado no XE3 (95,30MB), mas esse mesmo comportamento não é observado no segundo teste. O Update 1 do XE3 não traz melhorias no consumo de memória.

O servidor java apresentou um consumo elevado, mas esse resultado precisa ser avaliado com cuidado, pois se trata de uma medição realizada sobre a JVM. Aplicativos Java para servidores tendem a utilizar o máximo da memória disponível em prol do desempenho, mas esses aplicativos geralmente podem ser otimizados para utilizar diferentes quantidades de memória e isso não foi feito para esse servidor.

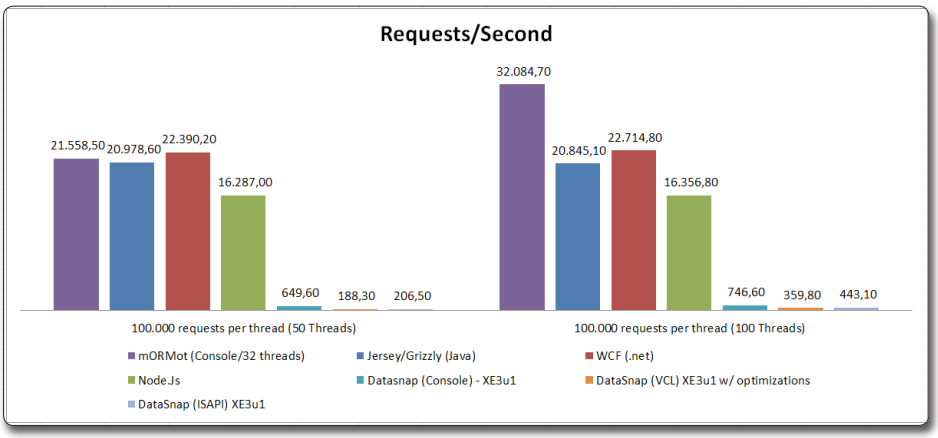


Figura 4. Teste de desempenho com multiplas threads

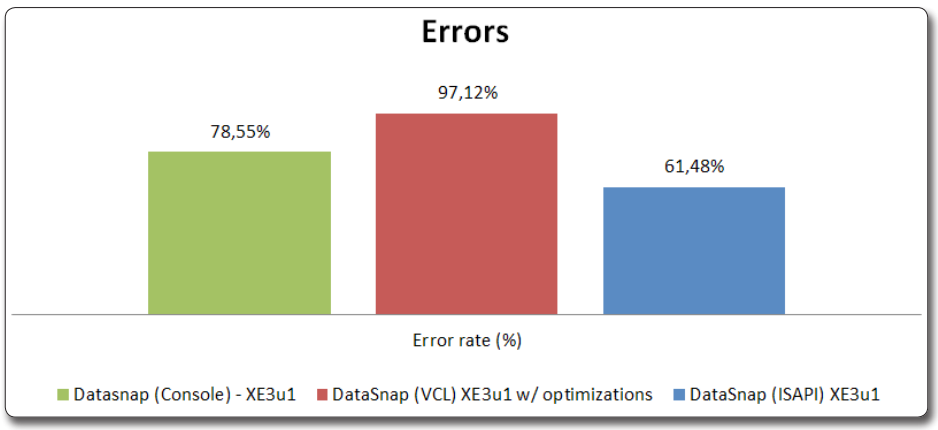


Figura 5. Porcentagem de requisições rejeitadas pelo servidor

Os demais servidores apresentaram consumo de memória baixo e invariável. Destaque para o mORMot que utilizou somente 6MB durante todo o teste. O consumo de memória invariável significa que independente das requisições que forem realizadas, a API sempre vai consumir a mesma quantidade de memória. Isso facilita o desenvolvimento com essa tecnologia porque não será necessário se preocupar com os picos de utilização de memória da API durante a execução do servidor.

Os testes sem concorrência apresentaram resultados a baixo do esperado para a tecnologia DataSnap. Ao que se referem à estabilidade, todas as APIs testadas apresentaram bons resultados nesse teste.

Testes com concorrência

Nesse teste a aplicação cliente possui diversas threads realizando requisições simultâneas. O servidor recebe dezenas de requisições ao mesmo tempo. Foram realizados testes com 50 e 100 threads buscando simular um ambiente com uma grande quantidade de usuários. Cada thread é responsável por enviar 100 mil requisições ao servidor. Como o hardware cliente possui um processador com quatro núcleos, não se pode considerar que são executadas 100 requisições exatamente ao mesmo tempo, algo que pode acontecer em um ambiente de produção. A **Figura 4** mostra os resultados do teste de desempenho de ambos os testes, com 50 e 100 threads.

No primeiro teste (50 threads) as tecnologias mORMot, Jersey e WCF apresentaram os melhores resultados. O Node.JS não obteve resultados tão bons, mas ainda assim impressiona devido a sua arquitetura que só aproveita um núcleo do processador. É possível ter vários servidores Node.JS na mesma estrutura para usufruir dos recursos de processadores com vários núcleos, o que traria resultados, provavelmente, superiores aos demais.

No segundo teste (100 threads) o mORMot obteve resultados significativamente melhores com relação aos demais, que praticamente repetiram o desempenho do primeiro teste.

O desempenho dos servidores DataSnap em ambos os testes é decepcionante. O servidor DataSnap leva 43 segundos para responder as requisições que um servidor mORMot responde em 1 segundo.

Avaliando os resultados tem-se a impressão de que as otimizações fazem o efeito contrário com relação a desempenho, mas como há uma variação muito grande entre as diferentes versões, não se pode afirmar isso com certeza.

A partir desses testes há uma nova variável a ser considerada, a quantidade de requisições rejeitadas pelo servidor. Não houve nenhuma requisição rejeitada nos testes sem concorrência, mas nos testes com concorrência flagramos um número considerável de erros, como mostra a **Figura 5**.

Esse índice aparece somente nos servidores DataSnap. Todos os servidores DataSnap apresentaram taxas de erros em níveis preocupantes, próximos da totalidade de requisições. Por mais que os problemas de estabilidade, que faziam o servidor travar estejam corrigidos na versão XE3 Update 1, um servidor que rejeita 97% das requisições não pode ser usado em produção. Mesmo o servidor ISAPI, que obteve o melhor índice, não poderia ser considerado um aplicativo estável com uma taxa de erros de 61%.

A **Figura 6** exibe os resultados dos testes de consumo de memória. O servidor Java mostrou novamente um consumo excessivo de memória, o que comprova que a tecnologia Java tende a usar o máximo de memória possível. Obviamente isso é um comportamento que pode ser alterado e não representa com fidelidade o potencial dessa tecnologia.

Servidores mORMot, WCF e Node.JS apresentaram novamente um consumo estático e muito baixo, idênticos aos resultados dos testes sem concorrência.

O servidor DataSnap sem otimizações consumiu uma quantidade grande de memória para este teste.

Percebe-se novamente que o consumo sobe conforme a quantidade de requisições aumenta. Se o teste fosse estendido, em menos de 24 horas o servidor teria chegado ao limite de memória disponível e entraria em colapso.

O resultado do servidor DataSnap com as otimizações é melhor, mas não deixa de ser preocupante. Esse servidor apresentou um consumo normal no primeiro teste (50 threads), embora tenha atingido o dobro do teste sem concorrência. Porém chegou a 250MB no segundo teste (100 Threads). Com as otimizações impedindo que sessões fiquem abertas, não deveria haver esse consumo elevado.

Comportamentos observados durante os testes

Nem todos os testes puderam ser acompanhados durante todo o tempo de sua execução, principalmente, devido ao tempo de execução dos testes com DataSnap, que levaram vários dias. Durante o monitoramento em tempo real de alguns testes notou-se alguns comportamentos estranhos, os quais serão apresentados a seguir. As **Figuras 7** e **8** mostram informações do servidor DataSnap com otimizações.

Nos gráficos do ProcessExplorer é possível perceber uma variação no consumo de memória e alguns picos nos índices de I/O. O I/O estava estável a 22.3KB e subitamente pulou para 464,5KB. Depois de algum tempo voltou a estabilizar. Ao mesmo tempo o servidor possuía 1153 threads. Foi possível observar mais de 1450 threads em determinados momentos. Não se encontrou uma explicação para esse comportamento, já que estamos usando um thread pool nesse servidor. Não deveria haver mais threads do que o informado no Thread Pool.

Outro comportamento estranho observado, este nos testes sem concorrência, é que alguns softwares interferem no desempenho do servidor DataSnap. Identificaram-se dois, Google Chrome e Eyebeam (VOIP). O mais impressionante é que esses softwares interferem positivamente no servidor, ou seja, o servidor fica mais

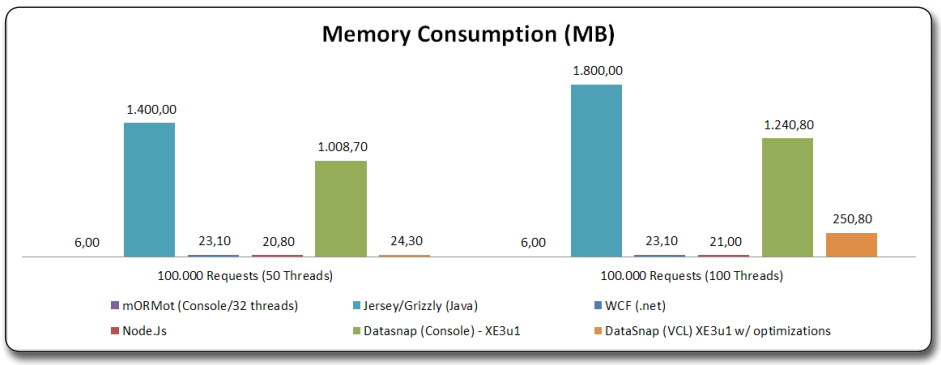


Figura 6. Consumo de memória em teste com múltiplas threads

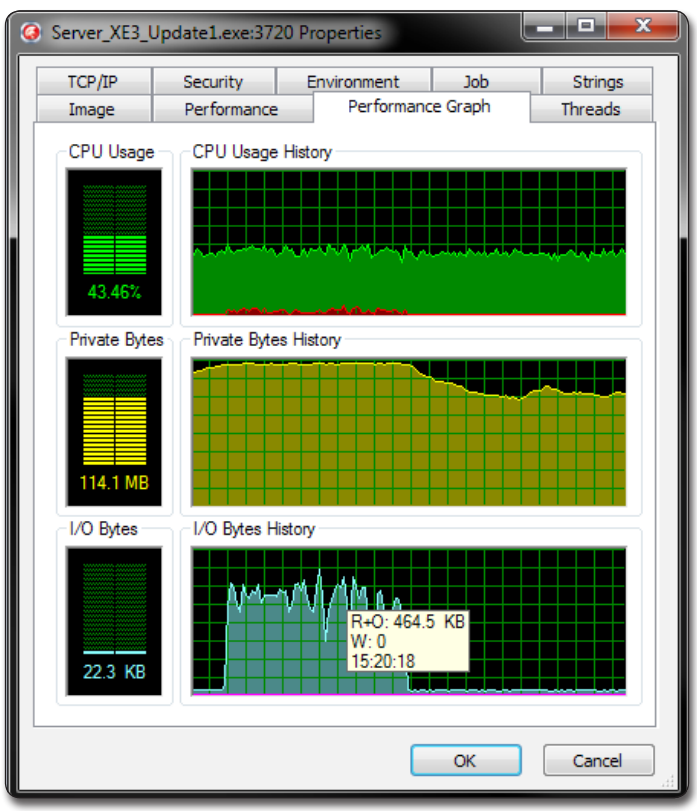


Figura 7. Servidor DataSnap durante teste com múltiplas threads

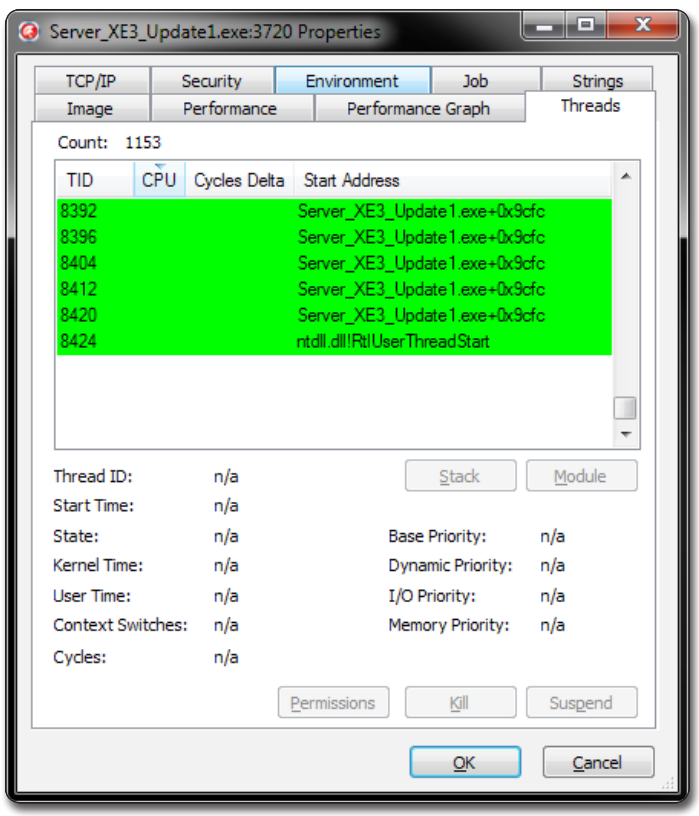


Figura 8. Servidor DataSnap durante teste com múltiplas threads

rápido quando esses softwares estão abertos. O servidor chega a ser quatro vezes mais rápido quando o EyeBeam está aberto. Já com o Google Chrome, o desempenho do servidor varia conforme você usa o navegador. A diferença é facilmente identificada monitorando os índices de I/O durante os testes.

Nota

Durante os testes aqui avaliados, as máquinas utilizadas não possuíam nenhum software que pudesse interferir no teste. Antivírus, Firewall, Google Chrome, Eyebem, etc.

Conclusão

O problema de estabilidade identificado na versão XE3, possivelmente também presente em versões anteriores, é um problema sério para quem pretende utilizar essa tecnologia e está utilizando essas versões do Delphi.

Quanto ao desempenho e estabilidade, o DataSnap não apresentou resultados satisfatórios no ambiente em questão. Isso não significa que ele não funcione bem em outros ambientes. Isso também não significa que um framework é melhor que o outro. Não se pode afirmar que um é melhor do que o outro somente avaliando desempenho e estabilidade, embora sejam itens importantes.

As melhorias lançadas no XE3 update 1 não representam uma grande reestruturação no DataSnap. Embora ele esteja um pouco mais estável, ainda não se pode esperar que a tecnologia funcione em ambientes com muita concorrência. Para resolver esses problemas a Embarcadero terá que reestruturar a API.

O Marco Cantù (atualmente Delphi Project Manager) escreveu a respeito do DataSnap em um de seus livros e a conclusão foi semelhante à apresentada por este artigo: "Eu penso que se você quer construir uma aplicação muito grande usando arquitetura REST você deve construir sua própria tecnologia ou usar algum desses protótipos. Para aplicações de pequeno e médio porte, por outro lado, você pode provavelmente se beneficiar do suporte nativo da tecnologia DataSnap." - Marco Cantù (Delphi Product Manager), Delphi 2010 Handbook – Tradução livre.

O DataSnap não é ideal para aplicações grandes, com uma grande quantidade de usuários, mas pode atender bem aplicativos de pequeno e médio porte. O mais importante é avaliar com cuidado cada framework, biblioteca ou componente antes de usar em qualquer projeto.

Os testes realizados foram testes sintéticos, visando o estresse máximo das APIs. Não representam com exatidão uma situação real. Aplicações reais dificilmente chegarão a esse nível de estresse na API, mas podem chegar próximo. Dependendo do tamanho e requisitos da aplicação você poderá utilizar o DataSnap sem problemas, mas também pode ser que você se veja obrigado a encontrar outra solução.

Roberto Schneiders
roberto.schneiders@gmail.com - robertocschneiders.wordpress.com/
Bacharel em sistemas de informação pela UNOESC-SMO. Atua como analista/programador de sistemas pela Sysmo Sistemas (www.sysmo.com.br). Co-fundador da Eletrone (facebook.com/EletroneBrasil). Trabalha com Delphi e Java.

Blog do Roberto Schneiders
http://robertocschneiders.wordpress.com/

Process Explorer
http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx

Artigo do Daniele Teti sobre DataSnap
http://www.danieleteti.it/2012/12/15/datasnap-concurrency-problems-and-update1/

Post do Marco Cantù sobre os problemas de desempenho
http://blog.marcocantu.com/blog/datasnap_deployment_performance.html

mORMot
http://synopse.info/fossil/wiki?name=SQLite3+Framework

WCF
http://msdn.microsoft.com/en-us/library/dd456779.aspx

Jersey
https://jersey.java.net/

Node.js
http://nodejs.org/

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Testes Unitários Automatizados

Utilizando DUnit para apoiar a agilidade

A Programação Extrema (ou somente XP) é uma metodologia ágil que se fundamenta em cinco valores básicos: Comunicação, Simplicidade, Feedback, Coragem e Respeito. Dentro desses valores vamos destacar três. A qualidade de um código fonte está diretamente ligada a sua simplicidade. Quanto mais complexo e mais poluído for o código fonte, mais trabalhosa é a sua manutenção e evolução. Um código fonte limpo é a melhor ferramenta para evolução de um software. Mas, como fazer um código fonte limpo?

Existem inúmeras maneiras que devem ser combinadas, como por exemplo, quebrar métodos grandes em métodos menores, quebrar classes grandes em classes menores, uso de padrões de projeto, boa nomenclatura de classes, métodos e variáveis, eliminação de duplicidade de código, etc. Manter um código fonte limpo não é uma tarefa fácil, exige um processo de refatoração (**Nota do DevMan 1**) constante.

Nota do DevMan 1

Martin Fowler define a refatoração como uma técnica disciplinada para a reestruturação de um corpo existente de código, alterando sua estrutura interna sem mudar seu comportamento externo. Cada refatoração é uma pequena transformação aplicada ao código-fonte, visando melhorar sua legibilidade, performance, organização, adequação a um padrão etc.

Alterar um mesmo código constantemente pode conduzir a um sério problema, a coragem. É realmente seguro reestruturar completamente um método de 40 linhas, quebrando em métodos menores e renomeando variáveis de escopo interno? E como ter certeza que está funcionando após a refatoração? É claro, a resposta óbvia é realizando uma bateria de testes. Mas ai vem a pergunta, como esses testes seriam executados? Manualmente? Quanto tempo esse processo demoraria? É garantido que serão testadas todas as possibilidades básicas do método? O feedback desses testes seria realmente confiável? Chegamos à conclusão que para

Resumo DevMan

Testes Unitários Automatizados:

Um dos maiores desafios no desenvolvimento de software é garantir a agilidade. Porém a palavra agilidade quando interpretada erroneamente pode causar consequências desastrosas. Por exemplo, em um cenário pontual, a solicitação de um novo recurso a um projeto, sem muito estudo, pode ser implementada e colocada em produção em questão de poucas horas. Mas, o que tornou essa implementação tão rápida? Esse questionamento inicial conduz a vários outros, por exemplo, já existiam bibliotecas prontas para essa funcionalidade? O software já estava prevendo essa funcionalidade antes mesmo de entrar em produção? Essa nova implementação não faz referência a outras rotinas, minimizando impactos? Ou a rotina não foi escrita da melhor maneira possível, não foi testada adequadamente e não foi refatorada? Na maioria das vezes a pergunta que recebe uma resposta é a última, de forma negativa. Não testar ou melhorar um código pode produzir com o tempo um bug nesse novo recurso. Mas isso é só o início do problema.

Outro desenvolvedor poderá realizar a correção, e com isso, como garantir que o recurso funcionará da maneira como foi planejado? Quanto tempo seria necessário para corrigir e testar esse recurso? Como garantir que o bug não irá voltar em produção? E se os requisitos dessa funcionalidade mudarem, será fácil a adaptação? Essas são questões pertinentes que são levantadas quando não existem testes suficientes, nem processos de apoio durante o desenvolvimento. Nesse artigo abordamos os testes unitários, suas características através do framework DUnit e sua aplicação prática.

Em que situação o tema é útil:

Códigos que são constantemente alterados necessitam de testes constantes. Rotinas importantes para o negócio desenvolvido também requerem testes constantes. Para acelerar o resultado desses testes são aplicadas automações, no caso, testes unitários. Dessa forma tem-se uma resposta rápida que diz se determinado código importante ainda funciona depois de uma alteração solicitada.

escrever um código fonte limpo e por consequência simples, é preciso estar constantemente executando refatorações, mas como criar coragem para executar a refatoração e como ter um resultado rápido dessa refatoração?

Testes unitários

Existem inúmeros tipos de testes, por exemplo, teste operacional, teste de volume, teste de stress, teste de configuração, teste de aceitação do usuário, etc. Grande parte desses testes foge ao controle do desenvolvedor, porém existe um tipo de teste que é de responsabilidade total do desenvolvedor, o teste unitário.

O teste unitário consiste na fase em que se testa pequenas unidades de software, por exemplo, os métodos de uma classe. Os testes unitários são testes criados e implementados pelo desenvolvedor, o que significa que esses serão rotinas automatizadas que não necessariamente necessitam de uma intervenção humana. O conceito inicialmente pode parecer complexo, mas imagine um método que calcule a soma de dois números, como mostra a **Listagem 1**.

Listagem 1. Método que realiza uma soma simples

```
function Somar(n1, n2: real):real;
var
  x : real;
  y : real;
Begin
  x := n1;
  y := n2;
  Result := x + y;
End;
```

O teste unitário desse método seria algo extremamente simples, como mostra o código a seguir:

```
Assert(Somar(2,2) = 4);
```

O método Assert testa se uma condição seja verdadeira, caso não seja, sobe uma exceção indicando onde o assert não foi atendido. Com isso já estamos, de forma automatizada, testando nosso método. Após qualquer alteração no método basta rodar a rotina que contém o Assert novamente e verificar se o método continua consistente. Feito nosso teste unitário, podemos sem medo refatorar esse método para torná-lo mais simples e mais claro, como mostra a **Listagem 2**.

Listagem 2 Método Somar refatorado

```
function Somar(APrimeiroNumero, ASegundoNumero : real):real;
begin
  Result := APrimeiroNumero + ASegundoNumero;
end;
```

Por mais ingênuo e simplista que seja esse exemplo, fica claro que com o teste unitário feito, o medo de alterar o código passou a ser muito menor. Dentro deste exemplo, imaginando que tivéssemos toda aplicação (ou boa parte dela) coberta por testes unitários não seria um problema o fato de o método Somar ser utilizado em diversas rotinas do software. Mas qual é o problema com o exemplo acima? Muito provável que após escrever o método Somar e executar seu teste (que provavelmente estava escrito em um formulário qualquer), esse código de teste é descartado, o que obrigaria escrevê-lo novamente caso o método sofra alguma alteração. Com certeza essa não é a melhor maneira de se fazer testes unitários em um software. Para isso existem frameworks especializados em testes unitários, como é o caso do xUnit.

Framework xUnit

Criado por Kent Beck, o xUnit inicialmente idealizado para SmallTalk (com o nome de SUnit) e popularizado posteriormente com sua versão para Java (com o nome de JUnit), consiste em um framework para geração e execução de testes unitários automatizados, que se consagrou como sendo o principal padrão de projeto de testes. O xUnit basicamente automatiza a rotina de execução de testes de uma forma simples e intuitiva tornando os testes vivos e evolutivos no mesmo ritmo do software.

Arquitetura

O framework xUnit possui em sua arquitetura os seguintes componentes básicos.

- Test Case - Qualquer classe de teste que o desenvolvedor crie deverá ser uma subclasse de Test Case;
- Test Suite - O xUnit executa suítes de testes, que nada mais são do que um conjunto de casos de testes;
- SetUp - Para cada método que será executado em nosso caso de testes, o método SetUp é executado antes. Geralmente o SetUp é utilizado parar criar a instância da classe que será testada;
- TearDown - Ao contrário do SetUp, o TearDown é executado ao final de cada método do caso de teste que está sendo executado, independente se o método do caso de teste seja executado com sucesso ou não. Geralmente usado para liberar da memória a instância criada no método SetUp.

Funcionamento

Através do conceito de reflexão (conhecido como RTTI no Delphi) o framework xUnit carrega todos os casos de teste registrados para execução, os executa (com ou sem interação do usuário) e provê feedback dos resultado obtidos na execução dos testes (seja visual ou através de arquivos).

Em sua interface visual, esses feedbacks consistem basicamente no padrão barra vermelha/barra verde. Mas, o que isso significa? É simples e intuitivo, quando um teste falhar este fica em vermelho, quando ocorre sucesso o teste fica verde. Nos testes que falharam é apresentada uma mensagem do tipo "o resultado esperado era <X> mas temos <Y>", entraremos nesses detalhes conhecendo o framework DUnit.

DUnit – A versão do xUnit para Delphi

Com o crescimento e popularização do xUnit, este framework

passou a ganhar versões para diferentes plataformas de desenvolvimento, como é o caso do framework DUnit para Delphi. Idealizado por Juancarlo Añez, o DUnit é um projeto open source que consiste praticamente em uma cópia fiel do JUnit, adaptada para Pascal. Nas versões mais novas do Delphi, o DUnit já vem como complemento.

Demonstração

Para demonstrar o uso da DUnit faremos uma classe de exemplo e um projeto de testes para testar essa classe. O intuito da nossa classe de exemplo será prover funções para lidar com strings.

Não faremos nada que o Delphi já não faça para nós. Criamos um projeto Win32 normal com o nome de ProjUtils e neste projeto criamos uma unit para a nossa classe (o nome da unit pode ser unt_str_utils), listada na **Listagem 3**.

Listagem 3. Declaração da classe base para demonstração dos testes unitários

```
unit unt_str_utils;

interface

type
 TStrUtils = class
 private
  FStr: String;
 public
  Constructor Create(const AStr : string);
  function Equals(const AStr : string):boolean;
  function EqualsIgnoreCase(const AStr : string):boolean;
  function ToUpper():string;
  function ToLower():string;
  property str : String read FStr write FStr;
 end;
```

A ideia é criar métodos simples e implementá-los de forma que possam ser melhorados, para que faça sentido a execução dos testes e a refatoração. A **Listagem 4** apresenta a implementação dos métodos da classe TStrUtils.

Como é possível de ver, simplesmente encapsulamos chamadas às funções e procedimentos existentes no Delphi para tratamento de strings.

Para testar essa classe, o mais comum seria criar um formulário VCL com um botão e um label e sair chamando os métodos públicos da classe, mas provavelmente estaríamos testando um método de cada vez e com certeza, após uma pequena bateria de testes iniciais, esse formulário seria descartado. Isso não é prático e de longe não é ágil. Vejamos o que o DUnit tem a nos oferecer.

Criando os testes

Sem fechar o projeto ProjUtils, crie um novo projeto de testes em File>New>Other >Unit Test>Test Project. Isso faz com que seja acionando o Wizard de criação de um novo projeto de testes, como apresentado na **Figura 1**. Este Wizard é divido em dois passos, no primeiro são feitas algumas configurações básicas, dentre elas, destacam-se três:

- **Source Project:** O projeto que será testado, entre outras palavras, o projeto que possui as classes que serão testadas através de casos de testes que serão criados;
- **Project Name:** É o nome do projeto de teste, o padrão é sempre o nome do projeto de origem mais a palavra Tests, exemplo ProjUtilsTests;
- **Location:** Onde o projeto de testes será salvo.

No segundo passo determinamos se a execução do teste será através de uma aplicação console ou através de uma aplicação visual (GUI).

Listagem 4. Implementação da classe TStrUtils

```
implementation
uses SysUtils;

constructor TStrUtils.Create(const AStr: string);
begin
 Self.FStr := AStr;
end;
function TStrUtils.Equals(const AStr: string): boolean;
begin
 result := AStr = Self.FStr;
end;
function TStrUtils.EqualsIgnoreCase(const AStr: string): boolean;
begin
 Result := UpperCase(AStr) = Self.ToUpper;
end;
function TStrUtils.ToLower: string;
begin
 Result := LowerCase(Self.FStr);
end;
function TStrUtils.ToUpper: string;
begin
 Result := UpperCase(Self.FStr);
end;

//explicar a classe
```

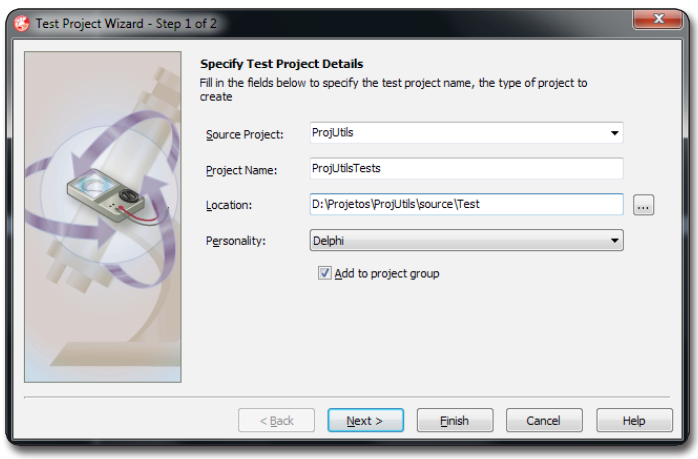


Figura 1. Wizard de criação de um projeto de testes

Com o projeto de testes criado, agora já é possível criar um caso de testes para nossa classe TStrUtils. Para isso, com o recém criado grupo de projeto (ProjUtils e ProjUtilsTests) aberto e

com o projeto ProjUtils marcado como projeto ativo, vamos em File>New>Other>Unit Test>Test Case.

Isso aciona o Wizard de criação de um novo caso de testes (**Figura 2**). Seguindo a simplicidade do primeiro, basta informar o Source File que iremos testar (no caso nossa unit unt_str_utils.pas) e escolher de quais classes e/ou métodos será criado um esqueleto de testes. Vale ressaltar aqui que esse assistente interpreta somente classes, sendo assim, records com conjunto de métodos não serão enxergados. No segundo passo desse assistente é informado o projeto de testes que esse caso de teste será adicionado e o nome da unit de testes. Por padrão, o nome da unit que será testada precedida da palavra Test, por exemplo, TestUnt_str_utils.pas.

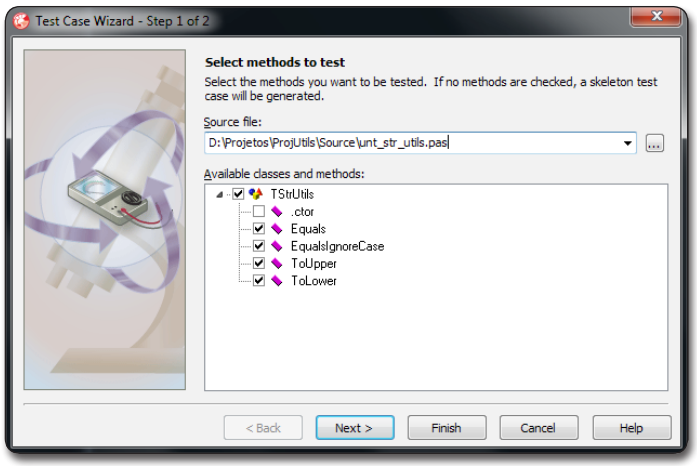


Figura 2. Wizard de criação de um caso de testes

Finalizando os passos desse wizard, será criada a unit de caso de testes com a seguinte estrutura vista na **Listagem 5**.

Implementando os testes

O caso de testes já está pronto, porém, obviamente ele não testa absolutamente nada, sendo assim, vamos entender e começar a dar vida a ele. Vamos começar por onde todos nossos testes vão começar, o método SetUp().

Como dito anteriormente, antes de cada teste o método SetUp() é executado, seguindo nosso exemplo, antes do teste TestTStrUtils.TestToUpper() ser executado, o método SetUp() será executado. Qual a vantagem disto? Vamos analisar o método SetUp(), criado pelo wizard da DUnit (**Listagem 6**).

Foi criada uma instância da classe TStrUtils, porém, levando em consideração a teoria do framework xUnit de que todo teste deve ser independente e não deve influenciar o resultado de outros testes, faz sentido que para cada teste seja criada uma nova instância do objeto a ser testado. Sendo assim, de maneira transparente o framework DUnit, antes de executar qualquer método, executa o método SetUp().

Sem mencionar que estamos evitando duplicidade de código, não sendo necessário instanciar o objeto em cada método de teste. Caso seja necessário além da criação da instância de um objeto

Listagem 5. Esqueleto de testes criado pelo wizard do DUnit

```
unit TestUnt_str_utils;
interface
uses
 TestFramework, unt_str_utils;
type
 TestTStrUtils = class(TTestCase)
 strict private
  FStrUtils: TStrUtils;
 public
  procedure SetUp; override;
  procedure TearDown; override;
 published
  procedure TestEquals;
  procedure TestEqualsIgnoreCase;
  procedure TestToUpper;
  procedure TestToLower;
 end;
implementation
procedure TestTStrUtils.SetUp;
begin
 FStrUtils := TStrUtils.Create;
end;
procedure TestTStrUtils.TearDown;
begin
 FStrUtils.Free;
 FStrUtils := nil;
end;
procedure TestTStrUtils.TestEquals;
var
 ReturnValue: Boolean;
 AStr: string;
begin
 ReturnValue := FStrUtils.Equals(AStr);
end;
procedure TestTStrUtils.TestEqualsIgnoreCase;
var
 ReturnValue: Boolean;
 AStr: string;
begin
 ReturnValue := FStrUtils.EqualsIgnoreCase(AStr);
end;
procedure TestTStrUtils.TestToUpper;
var
 ReturnValue: string;
begin
 ReturnValue := FStrUtils.ToUpper;
end;
procedure TestTStrUtils.TestToLower;
var
 ReturnValue: string;
begin
 ReturnValue := FStrUtils.ToLower;
end;
initialization
 RegisterTest(TestTStrUtils.Suite);
end.
```

Listagem 6. Método SetUp

```
procedure TestTStrUtils.SetUp;
begin
 FStrUtils := TStrUtils.Create;
end;
```

fazer outras parametrizações no objeto para que este seja útil, é no SetUp() que isso deve ser feito.

Para agilizar alguns testes (e também conseguir compilar nosso projeto de testes) vamos fazer uma pequena alteração no método SetUp() para que este fique aderente a nossa classe TStrUtils, como mostra a **Listagem 7**.

Listagem 7. Adaptação do método SetUp

```
procedure TestTStrUtils.SetUp;
begin
  FStrUtils := TStrUtils.Create('Testes DUnit');
end;
```

Já que para cada teste, uma instância do objeto a ser testado é criada, nada mais justo que a cada execução de teste, esse objeto seja liberado da memória. Este é o intuito do método TearDown(), segundo **Listagem 8**.

Sendo assim, o fluxo da execução do caso de teste seria desta forma algo como é mostrado na **Listagem 9**.

Listagem 8. Método TearDown

```
procedure TestTStrUtils.TearDown;
begin
  FStrUtils.Free;
  FStrUtils := nil;
end;
```

Listagem 9. Fluxo da execução dos métodos de teste

```
SetUp();
TestToUpper();
TearDown();

SetUp();
TestToLower();
TearDown();
```

Entendendo esse conceito básico, podemos escrever nosso primeiro teste unitário. Começaremos testando o método TStrUtils.Equals(). Esse método basicamente vai comparar se a string armazenada no atributo do objeto TStrUtils é igual a string passada como parâmetro para este método, retornando True para sucesso e False para falha. O teste básico para esse método poderia ser como mostra a **Listagem 10**.

Listagem 10. Teste unitário do método TStrUtils.Equals()

```
procedure TestTStrUtils.TestEquals;
var
  ReturnValue: Boolean;
begin
  ReturnValue := FStrUtils.Equals('Testes Dunit');
  CheckTrue(ReturnValue);
end;
```

Antes de avançarmos neste teste, vamos entender a presença de um novo elemento neste cenário, o **Check**. Assim como o Asserts, o Check basicamente avalia se uma condição seja verdadeira ou falsa, com a diferença que o Check é próprio do framework DUnit, sendo assim, caso uma condição não seja satisfeita, ele marca o teste como insucesso e permite a execução dos próximos testes, diferente do Asserts que interrompem a execução do código fonte. Neste teste foi utilizado o **CheckTrue**, porém, existe uma infinidade de outros Checks, sendo que alguns deles iremos explorar durante este estudo.

Voltando ao nosso teste, nota-se que este irá falhar (a letra "u" do DUnit está minúscula no teste), isso é bom, na verdade, isso é ótimo. Uma boa prática defendida pelos entusiastas dos testes automatizados é que o primeiro teste deve sempre falhar, essa é uma forma de garantir que seu teste é realmente confiável. Ao executar este teste, teremos uma barra vermelha (**Figura 3**).

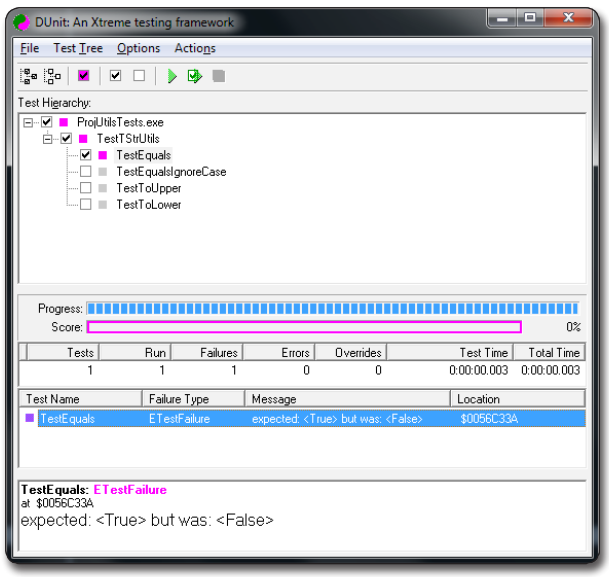


Figura 3. Executando o projeto de teste com uma falha

Para conseguirmos uma barra verde e nos certificarmos que nosso método Equals() funciona, vamos consertar nosso teste, como mostra a **Listagem 11**.

Listagem 11. Alteração no teste do método TStrUtils.Equals()

```
procedure TestTStrUtils.TestEquals;
var
  ReturnValue: Boolean;
begin
  ReturnValue := FStrUtils.Equals('Testes DUnit');
  CheckTrue(ReturnValue);
end;
```

Agora ao executarmos nossos testes teremos uma agradável e confortável barra verde, como mostra a **Figura 4.**

Note que estamos executando apenas o teste que estamos dando atenção no momento, isso porque, caso sejam executados os

outros testes da maneira como estão, todos dariam sucesso, pois em nenhum deles testamos alguma condição (utilizando o **Check**). Existe uma forma de inverter esse cenário. Nesta aplicação GUI da DUnit existe a opção Options>Fail TestCase if no Checks executed (em tradução livre, falhar o caso de teste se não forem executados checks), com essa opção marcada, todos os testes que não tiverem algum Check serão interpretados como falha.

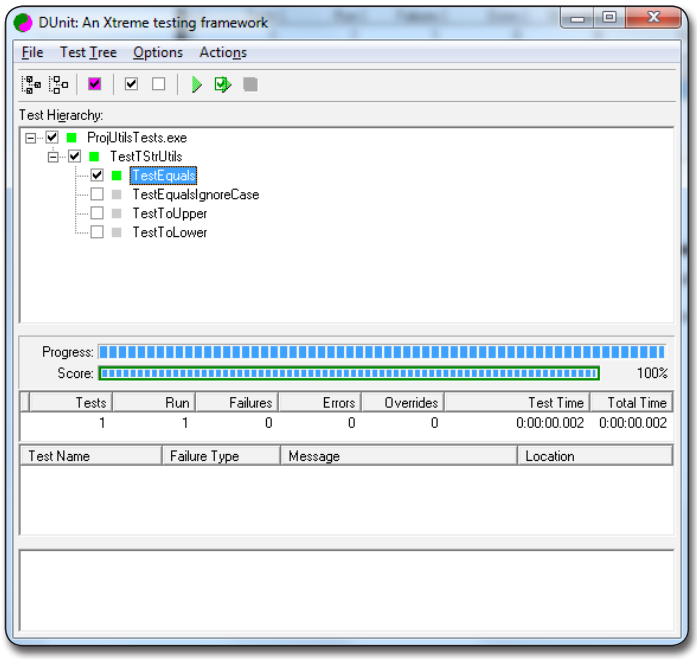


Figura 4. Teste funcionando

Com o método Equals() devidamente testado e funcionando, podemos partir para o teste do método EqualsIgnoreCase(). Esse método faz a mesma coisa que o método Equals(), com a diferença que ele ignora o fato do texto estar maiúsculo ou minúsculo, seu teste pode ser visto na **Listagem 12**.

Listagem 12. Teste unitário do método TStrUtils.EqualsIgnoreCase

```
procedure TestTStrUtils.TestEqualsIgnoreCase;
var
  ReturnValue: Boolean;
begin
  ReturnValue := FStrUtils.EqualsIgnoreCase('testes dunit');
  CheckTrue(ReturnValue);

  ReturnValue := FStrUtils.EqualsIgnoreCase('Uma string qualquer!');
  CheckFalse(ReturnValue);
end;
```

Desta vez estamos testando também se o método retorna False quando deveria retornar. Sem segredos ou surpresas, ao executar os testes, continuou com uma barra verde. Vale ressaltar aqui que, diferente de testes em formulários com botões, estamos executando todos os testes com apenas um clique e estamos obtendo feedback individualizado de cada teste.

Faremos um teste agora que nos fará repensar nessa classe (TStrUtils) como um todo. Vamos testar o método ToUpper(). Mas dessa vez, vamos incrementar um pouco o básico, como mostra a **Listagem 13**.

Listagem 13. Teste unitário do método TStrUtils.ToUpper

```
procedure TestTStrUtils.TestToUpper;
var
  ReturnValue: string;
begin
  ReturnValue := FStrUtils.ToUpper;
  CheckEquals('TESTES DUNIT',ReturnValue);

  FStrUtils.str := 'este é um teste com acentuação';
  ReturnValue := FStrUtils.ToUpper;
  CheckEquals('ESTE É UM TESTE COM ACENTUAÇÃO',ReturnValue,
  'Testes com acentuação');
end;
```

Antes de analisarmos este teste, vejamos o **CheckEquals** que é o Check mais comum do DUnit. Basicamente esse método (assim como a grande maioria de todos os checks) é composto dos seguintes argumentos:

- **Expected:** é o que esperamos que o método retorne;
- **Actual:** é o que o método realmente retornou;
- **Msg:** Uma mensagem que será apresentada caso o teste falhe.

Existem inúmeras sobrecargas desse método para serem utilizadas com diversos tipos de dados. Voltando ao nosso teste, aparentemente ele está ok, porém, ao executarmos, teremos uma triste barra vermelha (**Figura 5**).

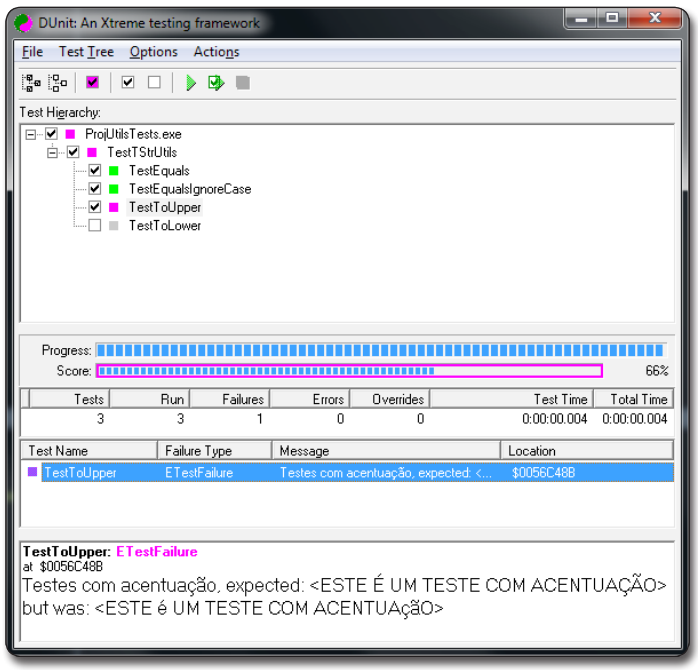


Figura 5. Barra vermelha

Ao ler a descrição da falha já fica mais claro o motivo da barra vermelha. O método ToUpper() não está considerando caracteres com acentuação. Isso é perfeito! Não o fato do método ToUpper() não estar funcionando, mas sim, o fato de termos encontrados a falha em desenvolvimento e não em produção. A resolução deste problema é algo muito simples, basta utilizar a versão do comando UpperCase que trata de acentos, a AnsiUpperCase (**Listagem 14**).

Listagem 14. Correção no método TStrUtils.ToUpper

```
function TStrUtils.ToUpper: string;
begin
  Result := AnsiUpperCase(Self.FStr);
end;
```

Pronto, barra verde novamente, mas essa barra vermelha anterior evidenciou uma fragilidade, nosso método EqualsIgnoreCase() funciona com strings que contenham acentos? Simples, vamos escrever um teste para verificar este cenário. Existem duas formas de criar esse novo teste, uma é testar esse cenário no teste já escrito e a outra forma é criar um teste novo, faremos da segunda forma para demonstrar que é possível evoluir a classe de caso de teste, como mostra a **Listagem 15**.

Listagem 15. Teste unitário para textos com acento

```
procedure TestTStrUtils.TestEqualsIgnoreCaseComAcento;
var
  ReturnValue : Boolean;
begin
  FStrUtils.str := 'este é um teste com acentuação';
  ReturnValue := FStrUtils.EqualsIgnoreCase
  ('Este é um Teste Com ACENTUAÇÃO');
  CheckTrue(ReturnValue,'Testes com acentuação');
end;
```

Ao executar esse novo teste, comprovamos a nossa suspeita, barra vermelha, mais uma vez, ótimo! Faremos agora a correção, como mostra a **Listagem 16**. Mais uma foi necessário utilizar o AnsiUpperCase.

Testes executados e uma barra verde como resultado significa que podemos partir para o próximo teste, o do método TStrUtils.ToLower(). Já sabendo a fragilidade com relação a acentos, faremos os testes já esperando este cenário (**Listagem 17**).

Ao executar esse teste, temos uma barra vermelha de resultado (Testes com acentuação, expected: <este é um teste com acentuação> but was: <este É um teste com acentuaÇÃo>), mas sem desanimar, basta realizar a já conhecida correção. No método TStrtUtils, no método ToLower fazemos uso da função AnsiLowerCase em substituição da LowerCase.

Pronto, toda nossa classe está coberta por testes e temos uma barra verde em todos os testes executados. Agora a classe está confiável o suficiente para entrar em produção.

Listagem 16. Correção do método EqualsIgnoreCase

```
function TStrUtils.EqualsIgnoreCase(const AStr: string): boolean;
begin
  Result := AnsiUpperCase(AStr) = Self.ToUpper;
end;
```

Listagem 17. Teste unitário do método TStrUtils.ToLower

```
procedure TestTStrUtils.TestToLower;
var
  ReturnValue: string;
begin
  ReturnValue := FStrUtils.ToLower;
  CheckEquals('testes dunit',ReturnValue);

  FStrUtils.str := 'ESTE É UM TESTE COM ACENTUAÇÃO';
  ReturnValue := FStrUtils.ToLower;
  CheckEquals('este é um teste com
  acentuação',ReturnValue,'Testes com acentuação');
end;
```

Alguns vão dizer "estou praticamente dobrando o tempo do meu desenvolvimento já que estou escrevendo muito mais linhas de código", mas não seria necessário testar essa classe de qualquer maneira antes de entrar em produção? E se o método EqualsIgnoreCase() tivesse a sua lógica alterada depois de anos em produção? Quanto tempo seria perdido primeiro para criar

coragem de alterar e segundo para se certificar que essas alterações não iriam impactar nos resultados esperados do método? Tenho certeza que com os testes automatizados essas perguntas não ficariam sem respostas. Sem mencionar que o tempo/custo de uma correção em ambiente de produção é infinitamente mais alto do que em ambiente de desenvolvimento.

Outros testes

Vamos imaginar agora que nossa classe TStrUtils deve ter um novo comportamento, toda vez que os métodos ToLower() e ToUpper() forem chamados, deverá subir uma exceção caso o atributo FStr esteja vazio. Essa exceção (**Nota do DevMan 2**) deverá ser do tipo EStrUtilsStringVazia.

Nota do DevMan 2

Pode-se dividir o conjunto de exceções de um sistema em dois subconjuntos: exceções tratadas, que são aquelas que são esperadas e até desejadas e que por serem conhecidas ocorrem dentro de um try .. except para prover uma mensagem amigável ao usuário e talvez até uma saída alternativa. É comum este try..except estar dentro de um try.. finally maior, para liberar recursos que de outra forma não poderiam ser liberados dentro de um except. A instrução finally sempre é executada, mesmo fechando-se o programa, caso o fluxo esteja dentro de um try, o finally será executado.

O outro grupo de exceções é o grupo das exceções não tratadas, onde a princípio não se sabe como e porque elas ocorrem, e talvez apenas o usuário saiba fazer a "mágica" acontecer, mas não saiba explicar como, e nem deveria, pois na concepção dele está fazendo tudo corretamente.

Faremos essa simples alteração, começando com a criação da exceção. Basta criar uma classe que herde da classe base Exception, como mostra o código a seguir:

```
EStrUtilsStringVazia = class(exception);
```

E subiremos essa exceção nos métodos em questão, como mostra a **Listagem 18**.

Listagem 18. Alterações nos métodos ToLower e ToUpper

```
function TStrUtils.ToLower: string;
begin
  if Self.FStr = EmptyStr then
    raise EStrUtilsStringVazia.Create('A String está vazia');
  Result := AnsiLowerCase(Self.FStr);
end;
function TStrUtils.ToUpper: string;
begin
  if Self.FStr = EmptyStr then
    raise EStrUtilsStringVazia.Create('A String está vazia');
  Result := AnsiUpperCase(Self.FStr);
end;
```

Feita esta alteração, a primeira coisa a se fazer é executar os testes unitários, que rodam sem problemas. Agora precisamos criar um teste para saber se esta nova lógica está realmente funcionando como o esperado. Para isso vamos utilizar um novo tipo de teste, como mostra a **Listagem 19**.

Listagem 19. Teste unitários para a exceção esperada

```
procedure TestTStrUtils.TestToUpperExceptionStringVazia;
begin
  FStrUtils.str := EmptyStr;

  ExpectedException := EStrUtilsStringVazia;

  FStrUtils.ToUpper();
end;
procedure TestTStrUtils.TestToLowerExceptionStringVazia;
begin
  FStrUtils.str := EmptyStr;

  ExpectedException := EStrUtilsStringVazia;

  FStrUtils.ToLower();
end;
```

Antes de executarmos estes testes, note que estamos usando um elemento novo, o **ExpectedException**. Como o nome já diz, nesta propriedade é possível indicar qual é a exceção esperada na execução do teste. Algo extremamente útil para quem costuma programar desta forma (fazendo o controle de falha com exceções e não com códigos de erro). Ao indicarmos que estamos esperando uma determinada exception no teste, caso durante a execução deste ocorra a exceção esperada, a DUnit vai interpretar que o teste passou com sucesso.

Conclusão

Ao executar os testes, temos uma gratificante barra verde. Porém algumas coisas no código da classe TStrUtils não estão cheirando bem. A duplicidade de código na geração da exceção de string vazia realmente não está boa, pense, se amanhã a frase padrão desta exceção passasse a ser "Esta rotina não deve ser utilizada para uma string vazia" ou, caso a exceção deixe de ser EStrUtilsStringVazia para ser qualquer outra, já seriam dois lugares diferentes para se alterar, isso porque estamos lidando com um exemplo extremamente simples. Qual seria a solução ideal? Refatorar esse código.

Para alguns essa palavra causa calafrios, frases do tipo "Se mexer no que está funcionando estraga" são realmente muito comuns. Mas, em nosso cenário não existem motivos para temer, nosso código está protegido pelos testes unitários. Caso a refatoração estrague algo, basta jogar a alteração fora e recomeçar novamente. Os testes nos dirão se a alteração está terminada e está segura. Então, basta isolar o código que verifica a string vazia em um método e fazer uso dele onde for necessário. Então, basta executar novamente os testes para termos certeza de que tudo continua funcionando como deveria.

A refatoração, assim como os testes unitários, é um processo de qualidade de software indispensável. Assim, finalizamos sobre testes unitários automatizados, que este sirva de base para entender e aplicar o conceito de uma forma simples e básica.

Um grande abraço e até a próxima

Diego Garcia
drgarcia1986@gmail.com
http://drgarcia1986.wordpress.com
@drgarcia1986
Analista/Desenvolvedor Delphi desde 2008, bacharel em Ciência da Computação e entusiasta de metodologias ágeis e engenharia de software. Atualmente escreve artigos sobre Delphi, Object Pascal e Firebird em seu blog e em publicações especializadas.

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Introdução à Biblioteca FireDAC

Conheça a nova biblioteca de componentes

O FireDAC pode ser considerado a edição da Embarcadero para a biblioteca de componentes de acesso a dados AnyDAC. Ele surge como uma forte opção no que se refere ao desenvolvimento de aplicação de banco de dados com Delphi, apresentando como pontos fortes sua facilidade de uso, alta performance e com recursos suficientes para atender as características gerais e particulares de cada SGBD suportado (**Nota do DevMan 1**).

Nota do DevMan 1

Atualmente, o FireDAC suporta os seguintes SGBDs:

- Advantage Database;
- IBM DB2;
- InterBase;
- Firebird;
- Microsoft Access;
- Microsoft SQL Server;
- MySQL;
- Oracle;
- PostgreSQL;
- SQLite;
- Sybase SQL Anywhere.

Atualmente o FireDAC é disponibilizado como um pacote singular que se integra ao Delphi. Para os usuários proprietários das edições Enterprise, Ultimate e Architect do XE3 (**Nota do DevMan 2**), ele é distribuído de forma gratuita, enquanto que para edições de menor calibre há um custo adicional. Fato é que nas futuras versões do Delphi o FireDAC já deverá acompanhar sua instalação, se tornando efetivamente parte integrante do IDE. Até lá, nada melhor do que já tirar proveito de sua atual disponibilidade, conhecendo todas as nuances que envolvem a utilização de suas classes, componentes e utilitários.

Resumo DevMan

Introdução à Biblioteca FireDAC:

Neste ano de 2013, dentre as recentes novidades anunciadas pela Embarcadero ao mundo Delphi, talvez a principal delas tenha sido a aquisição do AnyDAC, uma biblioteca de componentes de acesso a dados até então de propriedade da empresa DA-Soft Technologies. Conforme foi anunciado, o produto citado traz consigo uma nova gama de componentes para acesso aos dados de uma variedade de fontes, incluindo SGBDs (Ex: InterBase, Firebird, Oracle, MySQL, etc.), servidores de aplicação DataSnap, além de drivers ODBC e dbExpress. Ao ser incorporado ao Delphi, o AnyDAC foi rebatizado para FireDAC. Este então é o tema do presente artigo, cujo enfoque se dará em seu lado mais prático no desenvolvimento de aplicações Delphi. Vale ressaltar que, apesar da mudança de nome, o "core" do produto permaneceu intacto neste momento, fazendo com que a curva de aprendizado para quem já o utilizava fosse zero. Para os que ainda não o conhecem, este artigo tem a intenção de ser o ponto de partida para sua efetiva utilização.

Em que situação o tema é útil:

O acesso a dados é parte importante na construção de aplicações Delphi. Prova disso é que a ferramenta provê de forma nativa, uma série de opções para sua elaboração. Cada opção desta é refletida em uma biblioteca de componentes, ficando devidamente alocada na Tool Palette do IDE. Até o XE2, eram basicamente quatro as alternativas - BDE, dbExpress, InterBase Express e dbGo – no entanto, agora no XE3, este leque de opções ganha mais um elemento, o FireDAC que, de forma sucinta, pode ser definido como sendo uma biblioteca de acesso a dados universal.

Conectando-se a um banco de dados Firebird

O Firebird é sem sombra de dúvidas um dos SGBDs mais utilizados em aplicações de banco de dados Delphi. Seu baixo custo de manutenção, simplicidade de uso, boa performance e, fundamentalmente, seu caráter gratuito, são algumas das principais

características que o credenciam a tal posto. Ainda no Delphi 2010, em meio a suas novidades de lançamento, estava o novo driver dbExpress exclusivo para Firebird, que veio a atender um antigo e aclamado desejo da comunidade, que carecia de uma opção nativa ideal. Isto porque, até então, o desenvolvedor tinha basicamente duas alternativas para usar Firebird com dbExpress: recorrer a drivers de terceiro ou improvisar o driver para InterBase do dbExpress (o que causava certos transtornos em determinadas situações). Passado este momento, com a estabilização do driver citado, o Delphi ganha agora com o FireDAC, mais uma opção nativa para acesso a bancos de dados Firebird. Apesar de parecer algo natural e trivial num primeiro momento, o acesso a SGBD utilizando o componente de conexão (TADConnection) do FireDAC requer alguns tratamentos adicionais que, uma vez não realizados, podem causar certos transtornos ao desenvolvedor.

Nota do DevMan 2

Em sua atual versão (XE3), o Delphi, como produto, possui cinco edições:

- Starter Edition: edição de entrada, direcionada a estudos e construção de pequenos projetos. Por esta razão, não provê acesso a banco de dados e se restringe ao desenvolvimento para Win32;

- Professional Edition: complementa a edição Starter com recursos para acesso local de dados e compilação para Win32, Win64 e Mac OS X;

- Enterprise Edition: inclui todos os recursos da edição Professional, mais a parte de conectividade Client/Server e desenvolvimento de aplicações multicamadas (DataSnap);

- Ultimate Edition: inclui tudo da edição Enterprise, com o adicional do DB Power Studio, que provê ferramentas voltadas às tratativas de banco de dados, tais como Rapid SQL, DB Optimizer e DB Change Manager;

- Architect Edition: também inclui tudo da edição Enterprise, com o adicional do ER/Studio, uma ferramenta para modelagem e design de bancos de dados.

No TADConnection, uma definição padrão de conexão com um banco de dados envolve, basicamente, a configuração de duas de suas propriedades – DriverName e Params – num processo similar ao que é feito no TSQLConnection do dbExpress. Apenas para citar, em DriverName indica-se o nome do driver do FireDAC a ser utilizado, e em Params, os parâmetros da conexão (Ex: caminho do banco, usuário e senha do servidor, dialeto do banco, etc). O detalhe aqui fica por conta da ausência de um driver nominalmente específico ao Firebird, conforme mostra a **Figura 1**. Por esta razão, em um primeiro momento, é natural (e recomendada) a escolha pelo uso do driver IB que, em tese, visa atender tanto conexões para bancos InterBase quanto Firebird.

Tomando como exemplo a definição de uma conexão com o banco Employee, que acompanha a instalação do Firebird, o componente assume a configuração da **Listagem 1.**

Neste momento, ainda em tempo de Design, ao se tentar ativar a conexão um erro é gerado, tal como mostra a **Figura 2**.

Na busca por soluções para se corrigir tal situação, sem o devido conhecimento, fica fácil imaginar que não há muito a se fazer, uma vez que o driver e os parâmetros de conexão foram devidamente configurados. Todavia, o grande detalhe pode ser constatado no editor de conexão do próprio componente, que é acessado através de um duplo-clique sobre o mesmo. A janela que se abre possui quatro guias, dentre as quais está a de nome "Info". Nesta guia, são listadas em seções todas as informações relacionadas à conexão que está sendo estabelecida. Na seção "Client info" podem ser vistos então os detalhes relacionados à parte Client do banco, inclusive a biblioteca cliente que está sendo usada, tal como o exemplo a seguir, na **Listagem 2.**

Listagem 1. Configuração da conexão - componente ADConnection

```
01 object ADConnection1: TADConnection
02   Params.Strings = (
03     'DriverID=IB'
04     'Database=[caminho_instalação]\examples\empbuild\EMPLOYEE.FDB'
05     'User_Name=SYSDBA'
06     'Password=masterkey'
07     'Protocol=TCPIP'
08     'Server=localhost/3050')
09   LoginPrompt = False
10   ...
11 end
```

Listagem 2. Client Info

```
================================
01 Client info
================================
02 Loading driver IB ...
03 Brand = Interbase
04 Client version = 1100019900
05 Client DLL name =
  C:\Users\Public\Documents\InterBase\redist\InterBaseXE3\
  win32_togo\ibtogo.dll
```

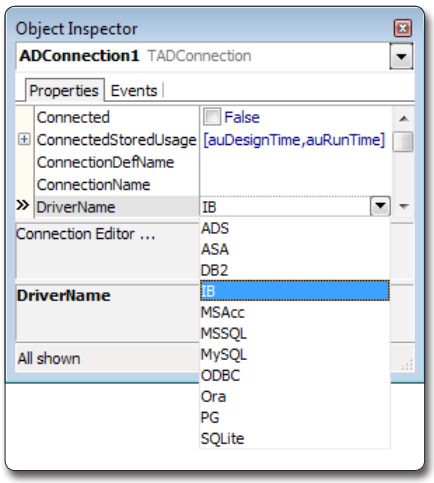


Figura 1. Lista de drivers para os bancos de dados suportados pelo FireDAC

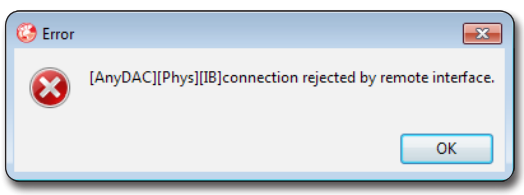


Figura 2. Erro gerado ao tentar se conectar a um banco de dados Firebird, utilizando FireDAC

Em "Client DLL name" fica nítida a "divergência" que está impossibilitando a estabilização da conexão com o banco de dados. Isto porque, por padrão, o FireDAC está tentando utilizar a biblioteca cliente ibtogo.dll do InterBase XE3, sendo que o correto, para este caso, seria a biblioteca fbclient.dll do próprio Firebird. Para que o FireDAC passe a utilizar a DLL cliente correta é necessário configurar seu arquivo de drivers denominado ADDrivers.ini, presente na pasta DB de sua instalação. De forma simples, este arquivo pode ser comparado ao dbxdrivers.ini do dbExpress.

Nativamente o arquivo ADDrivers.ini não traz conteúdo algum, dando mostras que toda a definição de drivers já parte de uma configuração padrão imposta pelo próprio framework. Todo e qualquer conteúdo adicionado a este arquivo irá então representar as customizações elaboradas pelo próprio desenvolvedor, mediante seu contexto de trabalho. No arquivo ADDrivers.ini há, basicamente, duas formas de se determinar a utilização da biblioteca correta por parte do driver. A primeira delas se faz simplesmente por sobrescrever a referida informação do próprio driver IB, tal como mostrado a na **Listagem 3**.

Listagem 3. Configuração do arquivo ADDrivers.ini

```
01 [IB]
02 VendorLib=C:\Program Files\Firebird\Firebird_2_5\bin\fbclient.dll
```

Simplesmente informamos ao seu parâmetro VendorLib o caminho completo da biblioteca cliente a ser utilizada. Apesar de funcional, esta forma restringe o driver IB a trabalhar apenas com bancos Firebird, uma vez que sobrescrevemos explicitamente seu parâmetro VendorLib. Isso impossibilitaria seu uso com bancos InterBase. Em vista dessa situação, a alternativa seria a definição de um novo driver virtual baseado no driver IB existente, mas com seu parâmetro VendorLib customizado para Firebird. Esta solução é então demonstrada na **Listagem 4**.

Listagem 4. Configuração para Interbase e Firebird

```
01 [FB]
02 BaseDriverID=IB
03 VendorLib=C:\Program Files\Firebird\Firebird_2_5\bin\fbclient.dll
```

Sendo assim, um novo driver virtual nomeado como FB foi definido, tendo como base o driver IB que é determinado no parâmetro BaseDriverID. Ao realizar este tipo de operação sobre o arquivo de drivers do FireDAC, uma boa prática é reiniciar o IDE certificando assim que o componente estará utilizando as informações mais recentes presentes no arquivo. Uma vez feito este procedimento o novo driver virtual é mostrado na lista de opções da propriedade DriverName, conforme mostra a **Figura 3**.

Uma vez realizado tais procedimentos, a conexão passará utilizar o driver do FireDAC já adequado a trabalhar com Firebird, fazendo uso da biblioteca cliente correta (**Listagem 5**).

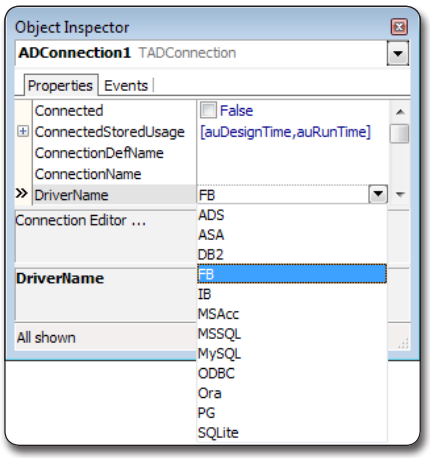


Figura 3. Novo driver virtual sendo mostrado na listagem de drivers do componente ADConnection

Listagem 5. FireDAC utilizando a biblioteca correta após configuração

```
================================
01 Client info
================================
02 Loading driver FB ...
03 Brand = Firebird
04 Client version = 205029900
05 Client DLL name = C:\Program Files\Firebird\Firebird_2_5\bin\fbclient.dll
```

Fazendo o deploy de aplicações FireDAC

No Delphi, as aplicações de banco de dados comumente requerem algum tipo de cuidado no que diz respeito ao seu deploy. Ao se utilizar dbExpress e TClientDataSet, por exemplo, é necessário distribuir junto da aplicação alguns arquivos adicionais, tais como a DLL do driver DBX utilizado, bem como a biblioteca MIDAS.dll. Com o FireDAC isso não é muito diferente, sendo necessária algumas ações complementares para que a aplicação possa ser devidamente implantada (deploy) em outros ambientes que não o de desenvolvimento.

A primeira ação ocorre ainda em tempo de Design. Num teste simples podemos tomar como ponto de partida a conexão anteriormente criada. De forma complementar, um componente DataSet do FireDAC (tal como um TADQuery) pode ser utilizado para se recuperar os dados da tabela Customer do banco de dados, para que sejam mostrados em um TDBGrid, conforme mostra a **Figura 4**. Até este momento, tudo está devidamente funcional. Todavia, ao se tentar executar a aplicação, uma Exception do FireDAC é disparada (**Figura 5**) interrompendo a sequência da execução.

Isto ocorre porque o FireDAC necessita de mais dois componentes (ou pelo menos da declaração de suas respectivas units) dentro do projeto. Como a própria mensagem sugere, o primeiro deles é um componente TADPhysIBDriverLink, que será o responsável pela conexão com a biblioteca cliente do banco de dados utilizado. Ele substitui a necessidade de se distribuir uma DLL externa do framework, tal como acontece no dbExpress.

O outro componente em questão é o do tipo TADGUIxWaitCursor, que provê o controle sobre o cursor de espera de operações. Sua ausência causa, em Runtime, a exceção mostrada na **Figura 6**.

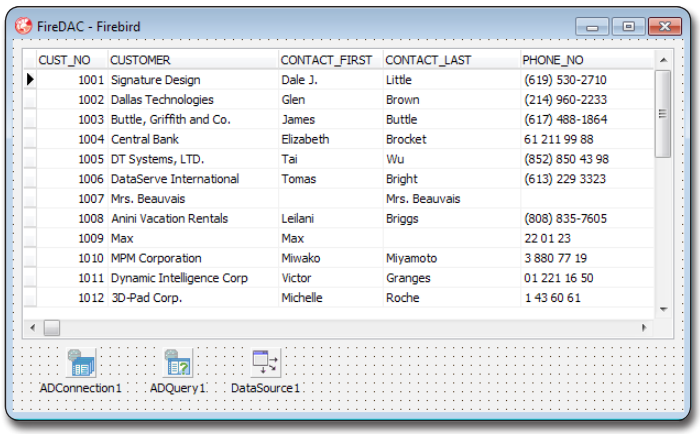


Figura 4. Aplicação de exemplo em tempo de desenvolvimento

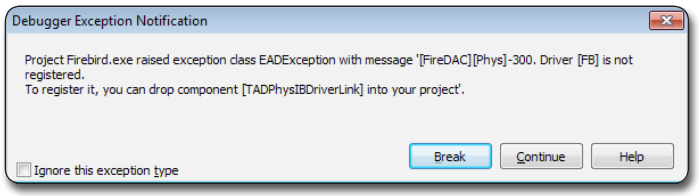


Figura 5. Exception do FireDAC

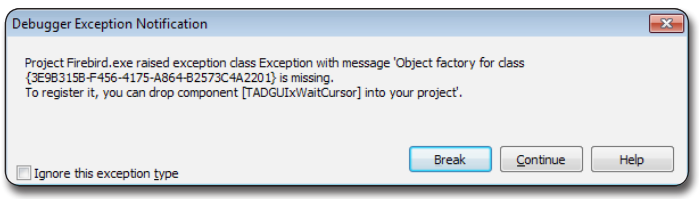


Figura 6. Exception do FireDAC

Por fim, pelo fato do acesso ao Firebird ter exigido algumas definições complementares no arquivo de drivers (ADDrivers.ini) do FireDAC, a presença deste se torna automaticamente obrigatória no deploy.

Operações com DataSets

Como não poderia deixar de ser, os DataSets providos pelo FireDAC oferecem recursos semelhantes aos providos por outros DataSets, tal como o TClientDataSet. Recursos estes que apesar de básicos, se tornam itens essenciais no desenvolvimento de qualquer aplicação de banco de dados no Delphi. Como forma de situar o leitor, a seguir são expostos, de forma prática, alguns destes recursos. Na medida do possível, será traçado um paralelo com outros recursos similares já providos de forma nativa pelo Delphi até então.

Ordenando registros em memória

De forma abrangente, todos os DataSets presentes na biblioteca do FireDAC proveem recursos para classificação local de dados. Isso significa dizer que, a partir de um conjunto de registros recuperados de uma fonte de dados, você pode ordená-los em memória num processo similar ao que é feito com ClientDataSet. Ainda com base nessa comparação, a forma padrão para se realizar a ordenação dos registros no FireDAC é também feita pelo uso da propriedade IndexFieldNames.

Tomando como base o exemplo anterior, onde foi utilizado um componente TADQuery, poderíamos centralizar a ação de ordenação de dados nos próprios títulos das colunas do DBGrid, que representam cada campo (Field) presente no DataSet.

Sendo assim, no evento OnTitleClick do controle, teríamos um código semelhante ao da **Listagem 6**.

Listagem 6. Realizando a ordenação do DBGrid

```
01 procedure TForm1.DBGrid1TitleClick(Column: TColumn);
02 begin
03   ADQuery1.IndexFieldNames := Column.FieldName;
04 end;
```

Aqui, a propriedade IndexFieldNames recebe o nome do Field que será usado na ordenação que, para este caso, trata-se do próprio Field associado à coluna clicada. Indo além, é possível ainda passar à propriedade não somente a um, mas múltiplos campos, tornando a classificação das informações ainda mais abrangente, tal como mostrado na **Listagem 7**.

Listagem 7. Ordenação de múltiplos campos

```
01 ADQuery1.IndexFieldNames := 'CAMPO_1;CAMPO_2;CAMPO_3;CAMPO_N';
```

No FireDAC, por padrão, esta ordenação é realizada de forma ascendente. Para que este comportamento seja alterado e a ordenação passe a ser descendente, basta apenas acrescentar a indicação ":D" junto ao nome do campo passado à propriedade IndexFieldNames. A seguir, na **Listagem 8**, é mostrado um exemplo a partir do mesmo evento citado anteriormente.

Listagem 8. Alterando a forma de ordenação crescente para descendente

```
01 procedure TForm1.DBGrid1TitleClick(Column: TColumn);
02 begin
03   ADQuery1.IndexFieldNames := Column.FieldName + ':D';
04 end;
```

Os DataSets do FireDAC possibilitam ainda outra forma de se estabelecer essa classificação de dados, que é através do uso de índices pré-definidos, armazenados em sua propriedade Indexes. A **Listagem 9** mostra um trecho de código do componente TADQuery, com um índice (de nome Ordenar Por Country e relacionado ao campo Country) definido em sua coleção Indexes.

Por fim, sua utilização se dá conforme a **Listagem 10**.

Listagem 9. Utilização de índices pré-definidos

```
01 object ADQuery1: TADQuery
02   ...
03   Indexes = <
04    item
05      Active = True
06      Name = 'Ordenar Por Country'
07      Fields = 'COUNTRY'
08    end>
09   ...
10 end
```

Listagem 10. Modo de uso do índice pré-definido

```
01 ADQuery1.IndexName := 'Ordenar Por Country';
```

Filtrando registros em memória

No ClientDataSet, dentre seus recursos mais usuais, está o de filtro que permite uma sub-seleção dos dados presentes em seu cache. Isso quer dizer que a filtragem de dados acontece de forma local, sem envolver qualquer tipo de conexão com o banco de dados, num aspecto totalmente distinto do uso de instruções SQL. Na **Listagem 11** é mostrado um código de exemplo que filtra os dados presentes em um ClientDataSet, através da passagem da condição de pesquisa, no formato string, para sua propriedade Filter. Em complemento, é atribuído o valor True à sua propriedade Filtered que, efetivamente, irá "filtrar" os dados em cache.

Listagem 11. Utilizando o filtro do ClientDataSet

```
01 ClientDataSet1.Filter := 'CUST_NO = 1001';
02 ClientDataSet1.Filtered := True;
```

De forma similar, os DataSets do FireDAC também fornecem este tipo de recurso. Num aspecto padrão, sua utilização se torna idêntica ao do ClientDataSet, conforme ilustrado na **Listagem 12**.

Listagem 12. Filtro em cache no FireDAC

```
01 ADQuery1.Filter := 'CUST_NO = 1010';
02 ADQuery1.Filtered := True;
```

Seguindo por esta linha, além dos filtros sobre valores de dados, o FireDAC possibilita ainda a filtragem sobre os status dos registros. Sendo assim, a partir do conjunto de dados de um DataSet, é possível selecionar, por exemplo, somente aqueles que estão marcados como "modificado", "incluído" ou "excluído" em cache. Tomando como base o componente TADQuery, por padrão, ele não mantém esse tipo de informação sobre os registros em cache, o que impossibilita o uso do recurso. Neste cenário, uma simples chamada ao seu método Post, por exemplo, faz com que toda e qualquer alteração realizada sobre um de seus registros seja imediatamente aplicada no banco. Ele não mantém um cache de atualizações, o que viabilizaria a aplicação de várias operações sobre o banco em um único lote, tal como acontece com ClientDataSet.

Para que o componente TADQuery passe a trabalhar com cache de atualizações, é necessário que sua propriedade CachedUpdates seja "ativada", ou seja, marcada como True. Isso irá fazer com que toda e qualquer atualização de dados passe a ser adiada para uma futura aplicação em lote no banco de dados, fazendo com que o componente sinalize o devido status a cada registro inserido, modificado ou excluído. Assim, ao chamar o método Post, por exemplo, as alterações são efetivadas somente em memória. Para que as mesmas sejam aplicadas no banco, é necessário uma chamada complementar ao método ApplyUpdates, num processo semelhante ao que é feito com ClientDataSet.

Uma vez que o CachedUpdates está ativado, é possível então se realizar filtros sobre os status dos registros, através da propriedade FilterChanges. A seguir, na **Listagem 13** é mostrado um exemplo simples de sua utilização, onde estão sendo filtrados os registros alterados, que ficam marcados no cache com o status de "modificado".

Listagem 13. Filtro de registros que sofreram alteração (em cache)

```
01 ADQuery1.FilterChanges := [rtModified];
```

Localizando registros

Em complemento ao uso de filtros, estão os métodos padrão para localização de registros em DataSets do FireDAC. Apesar de haver inúmeros métodos e formas para este fim, podemos inicialmente destacar dois deles, em razão de sua simplicidade e eficácia. O primeiro a ser mencionado é o Locate, já habitual de outros DataSets presentes no Delphi. Sua utilização aqui se dá de forma idêntica ao tradicional, tal como mostrado na **Listagem 14**.

Listagem 14. Locate com DataSets do FireDAC

```
01 ADQuery1.Locate('CUSTOMER', 'D', [loPartialKey, loCaseInsensitive]);
```

Nesta chamada, está sendo realizada uma busca com base no campo CUSTOMER, cujo valor seja iniciado com a letra D. Vale lembrar que o Locate efetivamente não retorna o valor do registro em si, apenas direciona o ponteiro para a primeira ocorrência encontrada.

Outro método de localização a ser citado é o LocateEx, que pode ser considerado uma evolução do Locate uma vez que permite também a utilização de expressões como critério de busca, e não somente simples valores. Na **Listagem 15** são mostradas duas situações de uso do LocateEx, cujo resultado será o mesmo do exemplo anterior, só que agora fazendo uso do método referido, através do uso de valores e expressão.

Trabalhando com campos calculados

Via de regra, o uso de DataSets em aplicações de banco de dados aumenta a gama de recursos disponíveis ao desenvolvedor, em meio ao processo de desenvolvimento. Dentre estes, estão os campos calculados, que são elementos virtuais definidos pelo próprio programador, visando atender alguma necessidade

pontual da aplicação. O grande diferencial dos campos calculados, sem dúvida alguma, está na flexibilidade da composição de seus valores. Em outras palavras, seus valores não possuem uma relação direta com o banco de dados, o que dá margens para uma ideal customização dessas informações. Num exemplo clássico, podemos ter um campo calculado para o Sub-Total de uma venda, cujo valor é o resultado da multiplicação dos valores das colunas Quantidade e Valor do Produto. Internamente, todo este processo é feito de maneira inteligente e rápida, todo em memória.

Listagem 15. Utilizando o LocateEx

```
// Baseado em valor
01 ADQuery1.LocateEx('CUSTOMER', 'D', [lxoPartialKey, lxoCaseInsensitive]);

// Baseado em expressão
02 ADQuery1.LocateEx('CUSTOMER like ''D%''', [lxoPartialKey,
   lxoCaseInsensitive]);
```

Nos DataSets do FireDAC, a forma mais tradicional de se definir campos calculados é diretamente em seu FieldsEditor. Dito isto, aproveitando o exemplo de uso da tabela Customer, podemos criar um campo do tipo Calculated que será a junção das informações de outros dois campos do DataSet: City e Country. A **Figura 7** exibe a janela de definição de um novo Field no DataSet, do tipo Calculated.

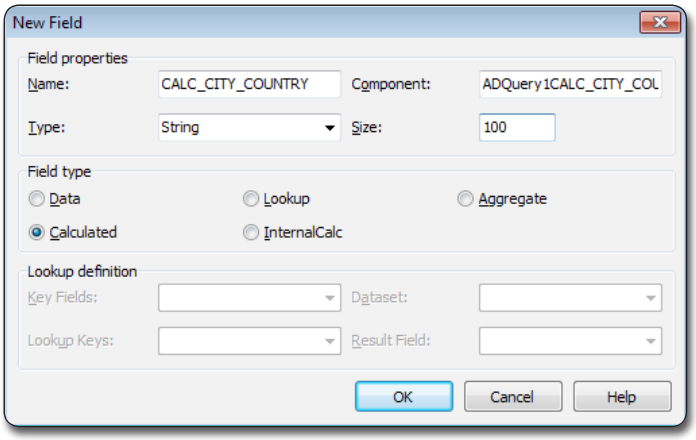


Figura 7. Criando campo calculado

Para o "cálculo" do valor do campo, os DataSets do FireDAC, assim como outros DataSets tradicionais do Delphi, possuem um evento exclusivo – OnCalcFields. Portanto, é nele que iremos escrever a instrução de formação do valor do campo, tal como a **Listagem 16**.

Listagem 16. Escrevendo campos calculados

```
01 procedure TForm1.ADQuery1CalcFields(DataSet: TDataSet);
02 begin
03   ADQuery1CALC_CITY_COUNTRY.AsString := ADQuery1CITY.AsString + ' - ' +
     ADQuery1COUNTRY.AsString;
04 end;
```

Apenas para comentar, no código, simplesmente atribuímos ao campo calculado criado o valor do campo CITY, concatenado com o valor do campo COUNTRY. O resultado disso pode ser visto em destaque na **Figura 8**.

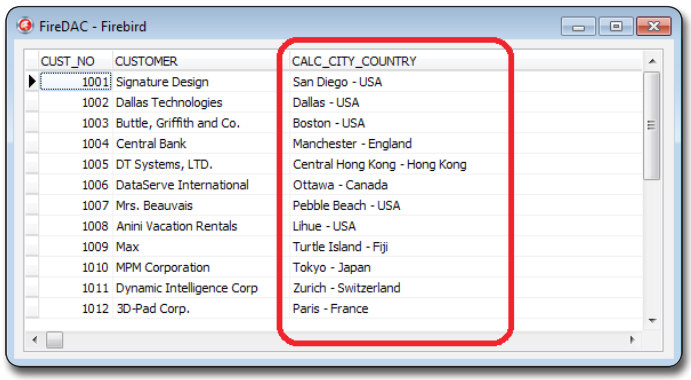


Figura 8. Criando campo calculado

Conforme pode ser visto, campos do tipo Calculated são definidos em nível de registro, ou seja, para cada linha do DataSet, seu valor é calculado. Em complemento a isso, é possível se definir campos calculados em nível de conjunto de registros, os quais são chamados de agregados (Aggregate Fields). Num exemplo trivial, podemos ter um campo deste tipo para sumarizar os valores de determinada coluna (preço, por exemplo), de todos os registros do DataSet. Sua definição básica envolve a criação de um novo campo do tipo Aggregate, aos mesmos moldes do que foi feito para o campo Calculated, bem como algumas configurações no DataSet (AggregatesActive = True) e nas propriedades do Field criado, conforme mostrado na **Listagem 17**.

Listagem 17. Configurando um campo Aggregate

```
01 object ADQuery1AG_COUNT_CUST_NO: TAggregateField
02   FieldName = 'AG_COUNT_CUST_NO'
03   Active = True
04   Expression = 'COUNT(CUST_NO)'
05 end
```

Neste trecho, FieldName representa o nome definido ao campo agregado, e a propriedade Expression contém a expressão a ser utilizada para a determinação de seu valor. Numa exemplificação simples, neste caso apenas está sendo feita uma contagem (COUNT) do campo CUST_NO, que irá resultar no total de registros presentes no DataSet, tal como um RecordCount.

Definindo um relacionamento Master-Detail com DataSets

Ainda citando as aplicações de banco de dados, é comum as situações onde se faz necessária a definição de um relacionamento mestre-detalhe entre os DataSets envolvidos. Olhando pelo lado nativo, a definição deste cenário em meio a utilização de componentes dbExpress e ClientDataSet já pode ser considerado algo natural ao desenvolvedor. Com a chegada do FireDAC,

apesar do cenário se manter o mesmo, os elementos envolvidos são outros. Sendo assim, torna-se importante o conhecimento de como conseguir os mesmos resultados através do uso de elementos desta nova biblioteca. A intenção aqui é fazer com que ao se apontar para um registro num DataSet "Master", todos os seus registros relacionados sejam automaticamente carregados em outro DataSet "Detail". Para uma exemplificação simples, podemos utilizar o próprio banco Employee do Firebird, aproveitando o relacionamento existente entre as tabelas Employee, que farão o papel de estrutura Mestre, e SalaryHistory, para a parte de Detalhe.

No FireDAC há basicamente duas formas de se definir este tipo de relacionamento entre seus DataSets: baseado em parâmetros ou baseado em série. O detalhe aqui é que ambas variam sobre como se dará a configuração do DataSet "Detail", permanecendo o DataSet "Master" inalterado em ambos os casos.

1º forma: Parameter based

Na forma baseada em parâmetro, a consulta SQL presente no DataSet "Detail" possui uma cláusula Where, onde seu campo de relacionamento com a estrutura "Master" (EMP_NO, neste caso) recebe o valor de um parâmetro de mesmo nome. No exemplo proposto, a instrução SQL ficaria como na **Listagem 18**.

Listagem 18. SQL de um mestre detalhe

```
01 select * from SALARY_HISTORY where EMP_NO = :EMP_NO
```

Para que o valor do parâmetro seja passado de forma automática do DataSet "Master" para o DataSet "Detail", é necessário que seja atribuído à sua propriedade MasterSource, um componente TDataSource devidamente ligado ao DataSet "Master". Isto é o suficiente para que o relacionamento seja estabelecido, tal como mostra a **Figura 9**.

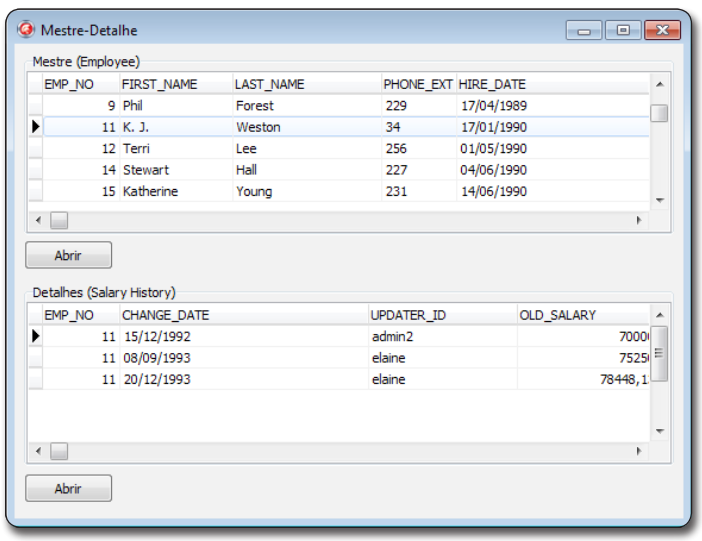


Figura 9. Relacionamento Mestre-Detalhe com FireDAC

2º forma: Range based

Já na forma baseada em série, os valores dos campos do DataSet "Master" são aplicados a uma série de registros do DataSet "Detail", que deverá possuir um índice ativo. Na prática, tem-se a mesma estrutura de componentes da forma anterior, mas com os seguintes diferenciais nas configurações do DataSet "Detail":

- A instrução SQL utilizada é mais abrangente, não dependendo de parâmetros explícitos. Assim, a instrução anteriormente usada seria algo como: "**select** * **from** SALARY_HISTORY";
- Em sua propriedade MasterFields deve-se indicar o campo do DataSet "Master" que será utilizado no relacionamento. Para o exemplo proposto, trata-se do campo EMP_NO;
- Conforme mencionado anteriormente, o DataSet "Detail" precisa de um índice ativo, relativo ao campo de relacionamento. Sendo assim, em sua propriedade IndexFieldNames, deve-se escolher o próprio campo EMP_NO de sua estrutura;

Por fim, em termos práticos, o resultado obtido deve ser o mesmo mostrado na **Figura 9**, ou seja, ao se navegar pelos registros da tabela Employee (Master), automaticamente vão sendo mostrados seus registros relacionados na tabela SalaryHistory. Posto isto, a escolha por uma ou outra abordagem é determinada pelas características e funcionalidades providas por cada uma, conforme listado a seguir.

Parameter based

- Permite se limitar o número de registros a serem retornados pela consulta SQL do DataSet "Detail";
- Os dados mantidos pelo DataSet "Detail" são mais atualizados;

Range based

- As operações realizadas sobre o DataSet "Master" são mais otimizadas, no que diz respeito à carga de trabalho sobre o SGBD;
- Permite se trabalhar com Cached Updates centralizado;
- Permite operar em modo "offline".

Definindo consultas SQL no QueryEditor do TADQuery

No FireDAC é comum o uso do componente TADQuery para as situações que envolvam consultas SQL a serem realizadas sobre um banco de dados. Conforme visto anteriormente, é nele que definimos, por exemplo, o comando "select" a ser utilizado numa estrutura de cadastro. Para tal, ele dispõe de uma propriedade denominada SQL, específica para o armazenamento dessa instrução. Por padrão, ao acioná-la, um simples editor de texto é aberto (**Figura 10**) para que o desenvolvedor escreva a instrução desejada. Apesar de útil, este recurso é bastante limitado. No pior das hipóteses, corremos o risco de digitarmos errado o comando SQL, causando um "crash" na aplicação, devido a um erro de sintaxe. Para melhorar este cenário, o TADQuery provê de um recurso adicional denominado QueryEditor que, entre outras coisas, permite testar a instrução que está sendo informada, bem como configurar os parâmetros envolvidos, caso hajam. Ele fica disponível no menu de contexto do próprio

componente e, uma vez acionado, sua janela é mostrada em tela (**Figura 11**).

Ao se definir uma consulta SQL no QueryEditor, ela já é automaticamente atribuída à propriedade SQL do componente. O detalhe aqui fica por conta da possibilidade de execução imediata da mesma, para testes de sintaxe e qualidade. Os dados retornados pela consulta são então exibidos na guia RecordSet, bem como a estrutura dos campos (ex: nome e tipo de dado) é mostrada na guia Structure (**Figura 12**).

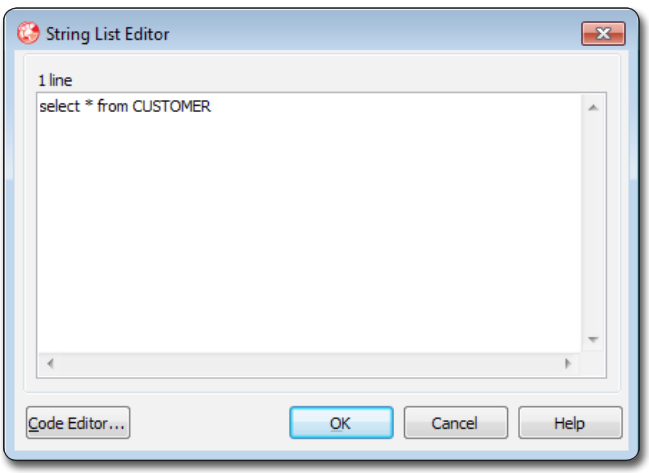


Figura 10. Editor SQL

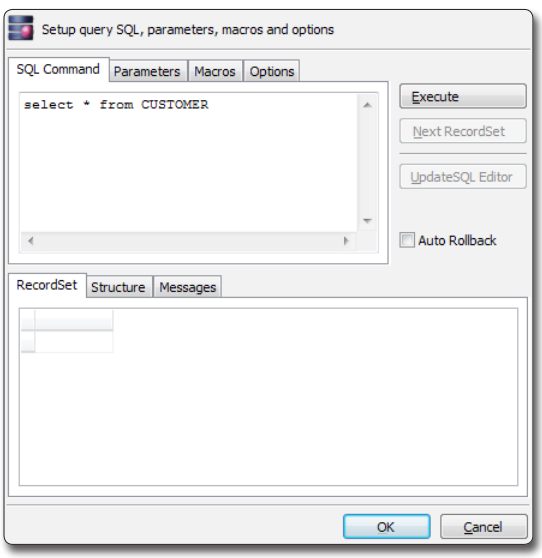


Figura 11. QueryEditor

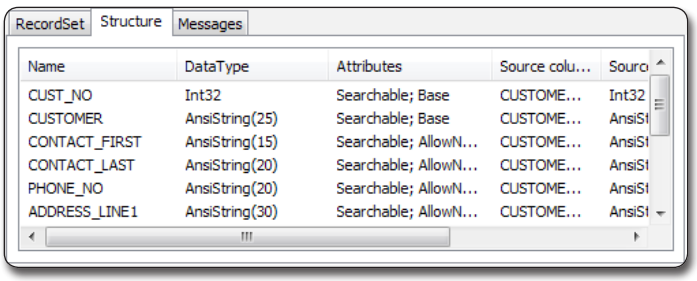


Figura 12. Estrutura de campos sendo mostrada no QueryEditor

Conclusão

Na comunidade, o reflexo imediato causado pelo anúncio da aquisição do AnyDAC (FireDAC) por parte da Embarcadero foi de muita contestação, argumentada pelo fato da já existência de uma solução nativa similar, que é o dbExpress. Polêmicas a parte, olhando pelo lado positivo, esta mais nova aquisição serve para certificar a constante evolução que a Embarcadero vem se propondo a realizar para o Delphi. A inclusão do FireDAC em seu contexto significa a abertura de novas possibilidades de desenvolvimento, bem como um aumento de seu já extenso quadro de recursos, no que diz respeito a acesso a dados.

Qual será a tendência daqui pra frente, dbExpress ou FireDAC? Como se dará a evolução de uma e outra biblioteca? Uma provável migração vale a pena? Inevitavelmente, essas e outras perguntas irão surgir neste momento, contudo, só o tempo é que poderá responder cada uma delas de forma efetiva. Pessoalmente falando não vejo motivos, por exemplo, para quem já tem toda uma bagagem feita em cima do dbExpress, almejar uma migração para FireDAC neste momento. Ambas as tecnologias dividirão terreno por um bom tempo ainda e a decisão por uma ou outra se dará, fundamentalmente, por questões de projeto e negócio.

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador Delphi há 8 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Delphi XE3 – Download Trial
https://downloads.embarcadero.com/free/delphi

Delphi XE3 – Página oficial do produto
http://www.embarcadero.com/products/delphi

FireDAC - Página oficial do produto
http://www.embarcadero.com/products/rad-studio/firedac

Embarcadero – Press Releases – Anúncio oficial da aquisição do AnyDAC
http://www.embarcadero.com/press-releases/embarcadero-technologies-acquires-high-performance-enterprise-database-connectivity-technology-for-developers

Embarcadero – InterBase XE3 – Página oficial do produto
http://www.embarcadero.com/products/interbase

Embarcadero – Interbase XE3 Developer Edition (Free) - Download
https://downloads.embarcadero.com/free/ibdeve

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Emissão de Boletos Bancários com ACBr

Utilização do ACBrBoleto para cobrança bancária

Resumo DevMan

Emissão de Boletos Bancários com ACBr:

Diante da atual necessidade do mercado comercial e industrial brasileiro, as aplicações que implementam funcionalidades que automatizam os processos mais comuns realizados pelas instituições e usuários, agilizando e simplificando todo o trabalho, evidentemente se destacam. Dentro deste contexto, contar com um software que realize a emissão e o controle de boletos é um diferencial que pode ser de muita relevância para o utilizador, uma vez que muito tempo pode ser necessário para a realização desta tarefa de uma forma não automatizada. Os desenvolvedores Delphi que desejam adicionar este e outros diversos recursos importantes em suas aplicações, contam com o conjunto de componentes do ACBr. Esta biblioteca é gratuita e está disponível há diversos anos, traz consigo várias classes, métodos, dentre outras funcionalidades, que permitem a implementação rápida e prática das mais variadas necessidades do mercado de automação comercial brasileiro, aumentando consideravelmente a produtividade e simplificando as fases de desenvolvimento e manutenção. Neste artigo inicialmente é apresentada uma visão geral da biblioteca ACBr e como realizar sua instalação, demonstrando posteriormente algumas das suas principais funcionalidades. Para concluir, é apresentado um simples exemplo prático de como empregar os componentes ACBrBoleto e ACBrBoletoFCFortes para automatizar todo o processo de emissão e controle de Boletos na aplicação.

Em que situação o tema é útil:

Este artigo apresenta uma visão geral dos componentes ACBrBoleto e ACBrBoletoFCFortes para que o desenvolvedor implemente de forma prática e rápida, a emissão e o controle de boletos em seu projeto, tornando assim sua aplicação comercial mais completa e atrativa, automatizando mais um processo para o usuário final.

O desenvolvimento de aplicações para o nicho Comercial e Industrial é sem dúvidas um dos mercados que recebem o maior foco quando o assunto é desenvolvimento de software. Diante deste fator, se destacam as organizações que oferecem soluções completas e interativas para seus utilizadores, cuja elaboração seja realizada no mais curto período de tempo. Sendo assim, contar com ferramentas de alta produtividade como o próprio ambiente de desenvolvimento RAD do Delphi, excelentes componentes adicionais e sólidos frameworks são um diferencial de suma importância e competitividade.

Presente há alguns anos, um recurso que se tornou um grande aliado dos desenvolvedores é o conjunto de componentes do projeto ACBr, cujo foco é a disponibilização de importantes funcionalidades para aplicações que atuam no mercado de automação comercial brasileiro. Esta biblioteca de ferramentas se tornou muito popular com o decorrer do tempo, além do mais, sua licença é gratuita e possui código fonte aberto, além de estar em constante atualização. Em meio a tantos equipamentos, compromissos fiscais, dentre outros, este conjunto de componentes apresenta soluções completas e sólidas para atender as mais variadas necessidades do projeto, fisco e mercado, facilitando muito a elaboração e manutenção das aplicações, visando sempre realizar o acesso direto aos recursos externos sem necessitar de funcionalidades de terceiros, como é o caso de DLLs (Dynamic Link Library) específicas.

Quanto aos recursos que acompanham a biblioteca, são basicamente componentes para a integração com impressoras fiscais (ECF – Emissor de Cupom Fiscal), porta serial, manipulação de impressoras de cheque, comunicação com balanças, funcionalidades de fala, geração de arquivos para o fisco, TEF (Transferência Eletrônica de Fundos), NF-e (Nota Fiscal Eletrônica), CT-e (Conhecimento de Transporte Eletrônico) e de uma forma mais especial neste artigo, o gerenciamento de Boletos, dentre muitos outros. Também é importante frisar que o ACBr está disponível para diversas versões do Delphi, C++ Builder e também o Lazarus, podendo ser empregado principalmente nos projetos baseados na VCL. Outro ponto muito útil desta biblioteca é com relação aos exemplos que acompanham a mesma, onde o desenvolvedor pode facilmente estudar e compreender como

empregar e configurar tais componentes em seus projetos. Além do mais, a equipe de desenvolvimento também pode contribuir para melhorias e correções nos pacotes que fazem parte do ACBr, simplesmente entrando em contato com os responsáveis pelo projeto, assim como realizar doações e implementar uma funcionalidade que ainda não está presente.

Com relação à emissão de boletos, esta é uma funcionalidade muito importante em uma aplicação comercial nos dias atuais, uma vez que automatiza e facilita o cotidiano do usuário final que necessita deste processo.

O processo de cobrança

De forma geral, o primeiro passo para a utilização deste e recurso é de fato que o usuário tenha uma conta bancária capacitada a receber esta funcionalidade, contando também com sua devida modalidade de cobrança, conhecida como carteira, que evidentemente são tarefas definidas em conjunto com a instituição bancária.

Dentro deste contexto, basicamente existem as carteiras registradas e as não registradas. Nas registradas, o banco precisa ser informado sobre todos os boletos que são emitidos, processo este que pode ser realizado através de um arquivo que recebe a denominação de "arquivo de remessa", sendo que neste caso o banco pode oferecer alguns serviços adicionais, tal como o protesto de títulos atrasados, e o arquivo de retorno, que possibilita ao usuário identificar os devidos pagamentos. Já nas carteiras não registradas, o banco não necessita ser previamente informado sobre os boletos que foram emitidos, sendo que o mesmo apenas toma conhecimento deste no momento em que ele é pago. Também é importante lembrar que cada banco possui sua taxa de cobrança para cada boleto emitido de acordo com o plano em questão contratado. Outra informação importante é o "Nosso Número", que por sua vez é um código único para a identificação do boleto em relação à conta utilizada, possibilitando a rápida identificação e a realização das operações, sendo que as características desta numeração também são definidas em conjunto com a instituição bancária.

Após a conclusão destas etapas, a aplicação que efetuará o gerenciamento dos boletos precisa passar pelo processo de homologação junto à instituição financeira, onde deverão ser gerados diversos boletos de teste e envios de arquivos de remessas para que o banco possa efetuar a aprovação e o usuário possa utilizar a aplicação. Para concluir, o desenvolvedor precisa tratar o arquivo de retorno disponibilizado pela instituição, verificando quais os boletos que foram recebidos e desta forma, efetuar a devida baixa na conta relacionada. O componente ACBrBoleto traz consigo diversos métodos para atender a estas necessidades, oferecendo suporte às muitas instituições bancárias da atualidade.

Download e instalação do conjunto de componentes ACBr

O ACBr pode ser facilmente obtido através do repositório web sourceforge.net (ver seção **Links**). O desenvolvedor pode efetuar seu download rapidamente utilizando uma ferramenta SVN (Apache Subversion - **Nota do DevMan 1**), por exemplo, o TortoiseSVN, simplesmente informando o endereço completo para o repositório. Feito o download, o conjunto de componentes do ACBr pode ser instalado basicamente de duas formas, utilizando a aplicação ACBrInstall ou através da compilação manual de cada pacote, levando em consideração uma determinada sequência. A instalação através do ACBrInstall é muito mais rápida e prática, onde o desenvolvedor pode definir quais as funcionalidades que deseja utilizar de uma forma bem simplificada. A **Figura 1** apresenta o ACBrInstall em execução com os pacotes desejados selecionados.

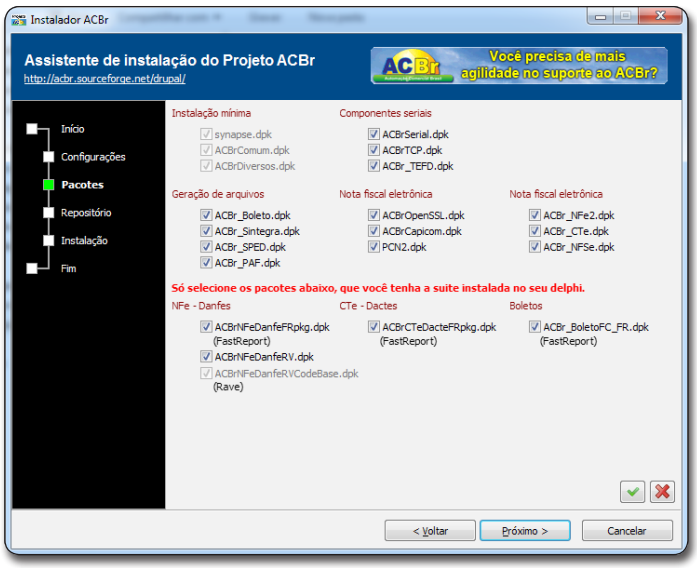


Figura 1. Janela de seleção de pacotes da ferramenta ACBrInstall

Nota do DevMan 1

O recurso SVN, Apache Subversion, é um sistema de controle de versão para arquivos cujo intuito é simplificar, centralizar e auxiliar o processo de desenvolvimento dos softwares, possibilitando seguramente que os desenvolvedores trabalhem no mesmo projeto de forma paralela e, em seguida, unam todas as tarefas desempenhadas em um único repositório, minimizando assim os conflitos e contratempos que podem ocorrer durante este tipo de processo. Uma aplicação gratuita muito popular dentro deste contexto é o TortoiseSVN, que traz consigo diversas funcionalidades úteis para a realização deste controle.

Entretanto, se preferir ou caso ocorram erros durante o processo de instalação automatizado, o desenvolvedor pode optar pelo procedimento manual. O primeiro passo é adicionar alguns diretórios que possuem os fontes da biblioteca no Library Path do Delphi (Tools>Options>Delphi Library>Library Path), sendo eles Fontes/ACBrComum, Fontes/ACBrSerial e os subdiretórios referentes ao conjunto de componentes que se deseja instalar, tal como ACBrBoleto. Realizado este processo, a próxima etapa é acessar o diretório que contém os pacotes para o Delphi (Pacotes>Delphi) e efetuar a compilação dos mesmos de acordo com a sequência apresentada a seguir:

synapse.dpk
ACBrComum.dpk
ACBrDiversos.dpk
ACBrSerial.dpk
ACBr_Boleto.dpk
ACBr_BoletoFC_Fortes.dpk

Além dos componentes básicos para o funcionamento do ACBr, também devem ser compilados os pacotes para a utilização dos recursos de Boleto, sendo eles o ACBr_Boleto, com as funcionalidades principais, e o ACBr_BoletoFC_Fortes, para possibilitar a impressão utilizando a ferramenta de elaboração de relatórios Fortes Report, lembrando que se preferir, o desenvolvedor pode compilar e utilizar o pacote ACBr_BoletoFC_FR para então empregar a ferramenta Fast Report ou ainda, o ACBr_BoletoFC_Quick para a impressão utilizando o tradicional Quick Report. Evidentemente, a ferramenta de relatório desejada já deve estar devidamente integrada e configurada no ambiente RAD, caso contrário exceções serão geradas e poderão surgir contratempos durante o processo de instalação da biblioteca. Contudo vale a pena lembrar que outros pacotes também podem ser compilados na sequência para a adição dos outros recursos que acompanham o ACBr, caso o desenvolvedor tenha intenção de utilizá-los.

Uma breve introdução aos componetes do pacote ACBr

Por ser uma biblioteca nacional, os diversos componentes que fazem parte do ACBr são facilmente configurados e interligados, além de serem muito intuitivos, uma vez que grande parte das suas propriedades e métodos estão em português.

Além disso, o desenvolvedor conta com uma diversidade de exemplos empregando os vários componentes para que possa ter uma visão geral de como empregá-los na sua aplicação. A **Tabela 1** apresenta alguns dos principais pacotes presentes nesta biblioteca.

Entretanto, o ACBr ainda conta com outros diversos componentes mais específicos, lembrando que os mesmos podem ser facilmente encontrados através do diretório "Pacotes".

Criando o banco de dados que será empregado neste exemplo

Para dar início a um exemplo prático de como utilizar o ACBrBoleto para todo o gerenciamento de boletos da aplicação, inicialmente será criado um novo banco de dados utilizando o SGBD Firebird, sendo que o mesmo recebeu a denominação de BOLETOS.FDB. Neste banco serão criadas duas tabelas, a primeira para que sejam armazenadas as contas a receber e, consequentemente, possibilitem a geração de boletos conforme os dados registrados, e a segunda, apenas para armazenar alguns dados. Na tabela de contas a receber, todas as informações foram centralizadas para facilitar no desenvolvimento do exemplo, mas em uma aplicação comercial, evidentemente convém separá-las em outras tabelas, como é o caso do cliente. A **Listagem 1** apresenta o código SQL que foi empregado para a elaboração da tabela de contas a receber.

Como é possível notar, esta é uma simples tabela que centraliza todos os dados da conta, possuindo como chave primária o campo CONREC_ID. O próximo passo é criar a tabela de configurações. A **Listagem 2** apresenta o código SQL utilizado para a elaboração da mesma.

Como poder ser notado, nesta tabela foi incluído o campo de número do arquivo de remessa para que o mesmo possa ser criado de forma sequencial e única. Neste exemplo, a tabela de configurações terá apenas um registro para controlar o número do arquivo de remessa, sendo assim o comando SQL INSERT INTO CONFIGURACOES VALUES (1,0) foi executado com esta finalidade.

Criando a aplicação e configurando a conexão com o banco de dados

Para dar início a um exemplo simples utilizando os componentes do ACBrBoleto, um novo projeto VCL deve ser criado (File>New>VCL Forms Application). Para realizar a conexão com o banco de dados criado anteriormente, será utilizado o framework dbExpress. Sendo assim, um objeto do tipo TSQLConnection deve ser adicionado ao formulário principal da aplicação para que este processo possa ser realizado, lembrando que se preferir, o desenvolvedor pode empregar outros frameworks, como é o caso do Zeos. Para realizar a devida conexão, as propriedades do

Pacote	Descrição
Synapse	Um pacote que deve ser apenas compilado que traz consigo os recursos necessários para que a comunicação com portas seriais possa ser realizada.
ACBrComum	Um conjunto de funcionalidades básicas para grande parte dos outros pacotes da biblioteca ACBr, trazendo consigo diversas Units importantes como é o caso da ACBrUtil, entre outras.
ACBrCapicom	Um conjunto de recursos para utilização da antiga biblioteca da Microsoft, CAPICOM, que por sua vez traz diversas funcionalidades para o emprego de criptografia e é uma das alternativas no desenvolvimento de soluções para a geração de Notas fiscais eletrônicas.
ACBrDiversos	Conta com uma variedade de componentes que podem ser úteis no dia-a-dia, como é o caso do ACBrBarCode para a geração de códigos de barra, e o ACBrExtenso para gerar valores em extenso.
ACBrSerial	Um pacote que contém diversos componentes que utilizam a comunicação Serial, como é o caso do ACBrBal e do ACBrECF.
ACBr_TEFD	Diversos recursos para o emprego de TEF nas aplicações.
ACBr_SPED	Contém funcionalidades para a geração dos arquivos SPED, contábil e fiscal.
ACBr_Boleto	Reúne os recursos necessários para a geração e o controle de boletos.
ACBr_BoletoFC_Fortes	Utilizado para possibilitar a impressão de boletos utilizando a ferramenta de elaboração de relatórios Fortes Report.
ACBrNFe2	Diversos componentes para a geração das Notas fiscais eletrônicas.

Tabela 1. Alguns dos principais pacotes da biblioteca ACBr

objeto inserido, SQLConnection1, devem ser definidas, conforme o trecho a seguir:

```
ConnectionName = FBConnection
Params.Database = BOLETOS.FDB
LoginPrompt = False
Connected = True
```

Listagem 1. Código SQL para a geração da tabela CONTASRECEBER

```
CREATE TABLE CONTASRECEBER (
 CONREC_ID INTEGER,
 CONREC_DATAEMISSAO DATE,
 CONREC_NOSSONUMERO VARCHAR(12),
 CONREC_NOME VARCHAR(50),
 CONREC_CNPJCPF VARCHAR(20),
 CONREC_ENDERECO VARCHAR(50),
 CONREC_NUMERO VARCHAR(10),
 CONREC_BAIRRO VARCHAR(50),
 CONREC_CIDADE VARCHAR(50),
 CONREC_ESTADO VARCHAR(2),
 CONREC_CEP VARCHAR(9),
 CONREC_DATAVENCIMENTO DATE,
 CONREC_PARCELA INTEGER,
 CONREC_VALOR NUMERIC(9,2),
 CONREC_DATAPAGAMENTO DATE,
 CONREC_VALORPAGAMENTO NUMERIC(9,2),
 CONREC_STATUS VARCHAR(1),
 CONSTRAINT PKY_CONTASRECEBER PRIMARY KEY (CONREC_ID)
);
```

Listagem 2. Código SQL para a geração da tabela CONFIGURACOES

```
CREATE TABLE CONFIGURACOES (
   CON_ID INTEGER,
   CON_NUMEROARQUIVOREMESSA INTEGER,
   CONSTRAINT PKY_CONFIGURACOES PRIMARY KEY (CON_ID)
);
```

Em ConnectionName é informado o tipo da conexão, neste caso FBConnection (Firebird), em Params.Database, o endereço da base de dados. Também não serão solicitadas informações para login de usuário no banco de dados (LoginPrompt = False) pois, já estão definidas em Params.User_Name/Params.Password. Para concluir, a conexão é ativada (Connected := True).

O passo seguinte para o exemplo é a adição de dois objetos do tipo TSimpleDataSet, também do framework dbExpress, cuja função é realizar um cache das informações na memória, conforme o comando SQL utilizado em sua propriedade DataSet. CommandText, possibilitando assim que as operações básicas de dados CRUD (Create, Read, Update and Delete) sejam executadas. O primeiro componente adicionado recebeu a denominação de sdsContasReceber, tendo sua propriedade Connection definida com o valor SQLConnection1. Também deve ser atribuído o comando SQL que será utilizado, neste caso SELECT * FROM CONTASRECEBER, lembrando que o mesmo deve ser definido na propriedade DataSet.CommandText. Vale a pena lembrar que, assim como outros componentes do tipo DataSet, o desenvolvedor pode adicionar os campos que desejar simplesmente efetuando um clique duplo sobre o mesmo e utilizar uma das opções presentes sem seu menu de contexto. Para finalizar a configuração do objeto sdsContasReceber, os eventos AfterDelete e AfterPost devem receber o comando ApplyUpdates para que todas as adições e alterações realizadas nas contas possam ser efetivadas em sua devida tabela, sendo assim, ambos eventos devem receber o trecho de código a seguir:

```
sdsContasReceber.ApplyUpdates(0);
```

Concluída esta etapa, o segundo componente TSimplesDataSet recebeu a denominação de sdsConfiguracoes, sendo o mesmo definido de forma semelhante ao objeto sdsContasReceber, porém com o comando SQL SELECT * FROM CONFIGURAÇÕES. Posteriormente, dois componentes do tipo TDataSource devem ser inseridos no formulário principal para que os componentes visuais possam ser interligados a suas devidas fontes de dados. O primeiro objeto DataSource adicionado foi renomeado para dsContasReceber e teve a propriedade DataSet definida com o valor sdsContasReceber, realizando assim a ligação com o mesmo, processo este que deve ser repetido com o segundo TDataSource, porém interligando este com o objeto sdsConfiguracoes.

O próximo passo é inserir os componentes visuais, onde neste exemplo foram adicionados inicialmente um objeto TPageControl, um TDBNavigator, quatro TSpeedButtons e os TDBEdits para os devidos campos da tabela de contas a receber, lembrando que o TDBNavigator deve ser ligado ao objeto sdsContasReceber através de sua propriedade DataSource.

Criada a interface gráfica da aplicação, devem ser adicionados ao formulário, os componentes responsáveis pela geração e impressão do boleto, TACBrBoleto e TACBrBoletoFCFortes. Neste momento, é necessário realizar a ligação entre ambos componentes simplesmente utilizando a propriedade ACBrBoletoFC do objeto ACBrBoleto1 e definido seu valor ACBrBoletoFCFortes1. Se preferir, o desenvolvedor pode configurar dados como banco e cedente diretamente através das propriedades do objeto ACBrBoleto1, entretanto, para possibilitar uma abordagem mais abrangente, como o emprego de diversos bancos por exemplo, este componente será configurado em tempo de execução, lembrando ainda que o desenvolvedor pode criar uma tabela de Bancos e assim carregar as informações dos mesmos para o componente de acordo com a escolha do cliente. Sendo assim, foi definido um novo método para realizar a configuração padrão para o objeto ACBrBoleto1. A **Listagem 3** apresenta o código empregado para realizar o carregamento das informações do banco e cedente em tempo de execução.

Inicialmente são definidas as informações padrões (linhas 4 a 6) como layout, logotipo, etc. Em seguida, são preenchidos os dados do banco, que neste exemplo é o Santander (linhas 7 a 11). Posteriormente, são atribuídas as informações do cedente, que por sua vez é o emissor do boleto. Tais informações foram preenchidas com dados fictícios, pois, evidentemente estas configurações devem ser definidas de acordo com os dados da empresa e do banco (linhas 12 a 19). Como pôde ser observado, por ser um componente nacional, as propriedades do ACBrBoleto são facilmente identificadas e preenchidas. Para finalizar esta etapa, este procedimento foi

adicionado ao evento OnClick do botão btConfigurarBoleto para que a configuração padrão seja carregada pelo próprio usuário antes de realizar a impressão do mesmo.

Listagem 3. Código do método ConfiguracaoPadrao

```
02 procedure TFrPrincipal.ConfiguracaoPadrao;
03 begin
04  ACBrBoleto1.ACBrBoletoFC.DirLogo                := '';
05  ACBrBoleto1.ACBrBoletoFC.Filtro                 := fiNenhum;
06  ACBrBoleto1.ACBrBoletoFC.LayOut                 := lPadrao;
07  ACBrBoleto1.Banco.Digito                        := 7;
08  ACBrBoleto1.Banco.Numero                        := 33;
09  ACBrBoleto1.Banco.Nome                          := 'Santander';
10  ACBrBoleto1.Banco.TipoCobranca                  := cobSantander;
11  ACBrBoleto1.Banco.TamanhoMaximoNossoNum := 12;
12  ACBrBoleto1.Cedente.Nome                        := 'CEDENTE DE TESTE';
13  ACBrBoleto1.Cedente.CNPJCPF                     := '';
14  ACBrBoleto1.Cedente.CodigoCedente               := '3000000';
15  ACBrBoleto1.Cedente.Agencia                     := '3000';
16  ACBrBoleto1.Cedente.AgenciaDigito               := '0';
17  ACBrBoleto1.Cedente.Conta                       := '1300000';
18  ACBrBoleto1.Cedente.ContaDigito                 := '0';
19  ACBrBoleto1.Cedente.UF                          := 'SP';
20 end;
```

Terminada a configuração básica da instituição bancária e cedente para a emissão dos boletos, o próximo passo é realizar o carregamento das informações da conta e logo em seguida, possibilitar a impressão do boleto. Para criar um novo título para o ACBrBoleto, o desenvolvedor deve criar um objeto do tipo TACBrTitulo e em seguida inicializá-lo devidamente. Sendo assim, o evento OnClick do outro botão adicionado anteriormente, renomeado para btImprirmirBoleto, deve receber um simples trecho de código para a realização desta tarefa. A **Listagem 4** demonstra o código do evento OnClick do botão btImprimirBoleto.

Como pode ser observado, é criado um novo título para o objeto ACBrBoleto1 através da função CriarTituloNaLista, sendo que o mesmo é atribuído à variável **Titulo**. Em seguida o objeto Titulo, que representa o que será cobrado no boleto, é preenchido com algumas informações padrões e com os dados presentes no dataset sdsContasReceber. Para concluir, o método Imprimir é chamado para que a janela de pré-visualização de impressão do boleto seja exibida, lembrando que neste exemplo a ferramenta empregada para esta tarefa é o FortesReport.

O passo seguinte é definir o código para a geração do arquivo de remessa. A **Listagem 5** demonstra o código que foi atribuído ao evento OnClick do botão btGerarRemessaSendo para realizar esta tarefa.

Neste trecho de código inicialmente é definido o diretório onde o arquivo será gravado através da propriedade DirArqRemessa do objeto ACBrBoleto1, que neste caso recebe o endereço atual da aplicação em execução, bem como o nome para o arquivo através da propriedade NomeArqRemessa. Posteriormente é verificado se existem boletos na lista para que a remessa possa ser gerada. Sendo esta condição satisfeita, o dataset sdsConfiguracoes é aberto e o número atual do arquivo de remessa é incrementado. Para concluir, o método GerarRemessa do objeto ACBrBoleto1 é chamado para criar o arquivo rapidamente, recebendo como parâmetro o número da remessa.

Listagem 4. Código do envento OnClick do botão btImprimirBoleto

```
01 procedure TFrPrincipal.btImprimirBoletoClick(Sender: TObject);
02 var
03  Titulo: TACBrTitulo;
04 begin
05  Titulo := ACBrBoleto1.CriarTituloNaLista;
06  Titulo.DataDocumento       :=
    sdsContasReceberCONREC_DATAEMISSAO.AsDateTime;
07  Titulo.LocalPagamento      := 'Pagável em qualquer banco
    até o vencimento';
08  Titulo.Carteira            := '101';
09  Titulo.EspecieDoc          := '01';
10  Titulo.NossoNumero         := sdsContasReceberCONREC_NOSSONUMERO.
    AsString;
11  Titulo.DataProcessamento   := Date;
12  Titulo.Aceite              := atNao;
13  Titulo.Sacado.NomeSacado := sdsContasReceberCONREC_NOME.AsString;
14  Titulo.Sacado.CNPJCPF      := sdsContasReceberCONREC_CNPJCPF.AsString;
15  Titulo.Sacado.Logradouro   := sdsContasReceberCONREC_ENDERECO.AsString;
16  Titulo.Sacado.Numero       := sdsContasReceberCONREC_NUMERO.AsString;
17  Titulo.Sacado.Bairro       := sdsContasReceberCONREC_BAIRRO.AsString;
18  Titulo.Sacado.Cidade       := sdsContasReceberCONREC_CIDADE.AsString;
19  Titulo.Sacado.UF           := sdsContasReceberCONREC_ESTADO.AsString;
20  Titulo.Sacado.CEP          := sdsContasReceberCONREC_CEP.AsString;
21  Titulo.Vencimento          :=
    sdsContasReceberCONREC_DATAVENCIMENTO.AsDateTime;
22  Titulo.Parcela             := sdsContasReceberCONREC_PARCELA.AsInteger;
23  Titulo.ValorDocumento      := sdsContasReceberCONREC_VALOR.AsFloat;
24  Titulo.ValorAbatimento     := 0;
25  Titulo.ValorDesconto       := 0;
26  Titulo.ValorMoraJuros      := 0;
27  Titulo.DataProtesto        :=
    sdsContasReceberCONREC_DATAVENCIMENTO.AsDateTime + 5;
28  Titulo.OcorrenciaOriginal.Tipo := toRemessaBaixar;
29  Titulo.Instrucao1          := 'Não cobrar juros de mora';
30  Titulo.Instrucao2          := 'Protestar no 5 dia útil após o vencimento';
31  ACBrBoleto1.Imprimir;
32 end;
```

Listagem 5. Código do envento OnClick do botão btGerarRemessa

```
procedure TFrPrincipal.btGerarRemessaClick(Sender: TObject);
begin
 ACBrBoleto1.DirArqRemessa      := ExtractFileDir(Application.ExeName);
 ACBrBoleto1.NomeArqRemessa := 'Remessa.txt';
 try
  if ACBrBoleto1.ListadeBoletos.Count > 0 then
  begin
   sdsConfiguracoes.Open;
   sdsConfiguracoes.Edit;
   sdsConfiguracoesCON_NUMEROARQUIVOREMESSA.AsInteger :=
   sdsConfiguracoesCON_NUMEROARQUIVOREMESSA.AsInteger + 1;
   sdsConfiguracoes.Post;
   ACBrBoleto1.GerarRemessa
   (sdsConfiguracoesCON_NUMEROARQUIVOREMESSA.AsInteger);
  end;
 except
  ShowMessage('Houve um erro ao gerar a remessa!');
 end;
end;
```

O passo final para este projeto é definir o código para a leitura do arquivo de retorno, disponibilizado pelo banco para que o usuário possa carregá-lo no sistema e o componente ACBrBoleto realize sua leitura e carregamento. A **Listagem 6** apresenta o código que foi definido no botão btProcessarRetorno para realizar esta tarefa.

Inicialmente é criado um novo objeto do tipo TOpenDialog (linha 7), lembrando que este componente possibilita a utilização da janela

de diálogo padrão para a abertura de arquivos. Posteriormente, a mesma é executada e o endereço do arquivo selecionado pelo usuário é armazenado na variável Arquivo (linha 9). Se tal variável estiver preenchida (linha 10), então é realizada a configuração padrão para o objeto ACBrBoleto1, bem como é definido o endereço para que o mesmo possa ler e carregar o arquivo através do método LerRetorno (linha 14). O total de títulos carregados é então atribuído para a variável TotalBoletos (linha 15). Em seguida, é realizado um laço for para varrer todos os boletos e verificar se o valor recebido do titulo atual é maior que zero. Sendo satisfeita esta condição, então o método Locate do objeto sdsContasReceber é chamado para localizar o registro referente ao boleto atual utilizando o valor do campo nosso número (linha 20). Caso o registro seja encontrado, então a data de pagamento, o valor e o status da conta a receber são atualizados. Para concluir, o objeto criado em tempo de execução, OpenDialog, é liberado da memória através do método FreeAndNil.

Listagem 6. Código do envento OnClick do botão btProcessarRetorno

```
01 procedure TFrPrincipal.btProcessarRetornoClick(Sender: TObject);
02 var
03  Arquivo: string;
04  OpenDialog: TOpenDialog;
05  TotalBoletos, i: Integer;
06 begin
07  OpenDialog := TOpenDialog.Create(Self);
08  OpenDialog.Execute;
09  Arquivo := OpenDialog.FileName;
10  if Length(Trim(Arquivo)) > 0 then
11  begin
12   ConfiguracaoPadrao;
13   ACBrBoleto1.NomeArqRetorno := Arquivo;
14   ACBrBoleto1.LerRetorno;
15   TotalBoletos := ACBrBoleto1.ListadeBoletos.Count;
16   for i := 0 to TotalBoletos - 1 do
17   begin
18    if ACBrBoleto1.ListadeBoletos.Objects[i].ValorRecebido > 0 then
19    begin
20     if sdsContasReceber.Locate('CONREC_NOSSONUMERO',
       ACBrBoleto1.ListadeBoletos.Objects[i].NossoNumero, []) then
21     begin
22      sdsContasReceber.Edit;
23      sdsContasReceberCONREC_DATAPAGAMENTO.AsDateTime :=
        ACBrBoleto1.ListadeBoletos.Objects[i].DataBaixa;
24      sdsContasReceberCONREC_VALORPAGAMENTO.AsFloat  :=
        ACBrBoleto1.ListadeBoletos.Objects[i].ValorRecebido;
25      sdsContasReceberCONREC_STATUS.AsString := 'R';
26      sdsContasReceber.Post;
27     end;
28    end;
29   end;
30 end;
31 FreeAndNil(OpenDialog);
32 end;
```

Neste momento, o exemplo para utilização de boletos está concluído. A **Figura 2** apresenta a visualização de um boleto fictício elaborado através desta aplicação.

Conclusão

Neste artigo foi possível ter uma visão geral sobre o componente ACBrBoleto, que por sua vez pode ser empregado na rápida geração de boletos, criação de arquivos de remessa e manipulação de arquivos de retorno, adicionando a aplicação em questão, uma funcionalidade muito importante que automatiza os processos realizados pelos usuários finais. Como visto, bastam algumas configurações, pequenos trechos de códigos e a aplicação já está hábil para emitir e gerenciar os boletos, restando apenas a fase de homologação em conjunto com a instituição financeira.

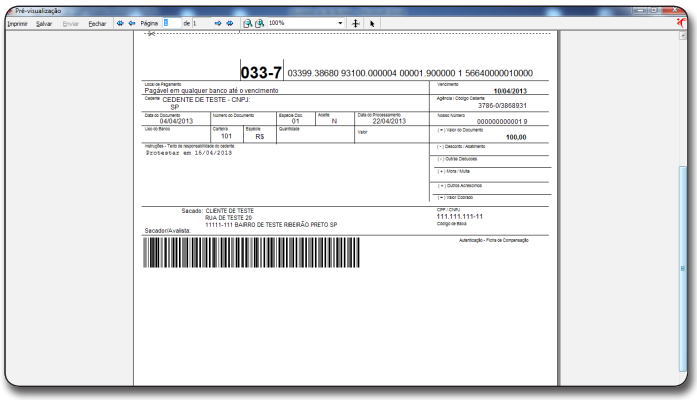


Figura 2. Boleto elaborado na aplicação de exemplo utilizando o ACBrBoleto

Também é importante lembrar que o ACBr possui outros recursos de extrema relevância quando o assunto é automação comercial, tal como toda a manipulação de notas fiscais eletrônicas. De forma geral, o ACBr é um grande aliado para agilizar e simplificar o desenvolvimento das soluções cotidianas para atender as exigências do mercado nacional, além do mais o mesmo é uma biblioteca que está em constante expansão, recebendo várias atualizações e em alguns casos, correções, sem contar que já está a um bom tempo auxiliando o dia a dia da equipe de desenvolvimento.

Giuliano Scombatti Pinto
giuliano@sygnux.com.br
É Analista Desenvolvedor da Sygnux Software (www.sygnux.com.br) localizada em Monte Alto/SP, atuando também com desenvolvimento Web e Android. Formado pela Fatec/TQ, começou a desenvolver com Delphi em 2003. Atualmente trabalha com Delphi, Java, PHP e Flash/Flex. Professor de Informática de Projetos Sociais da Prefeitura Municipal de Monte Alto/SP.

Maiores informações e download da biblioteca ACBr
http://acbr.sourceforge.net/drupal/?q=node/37

Recursos adicionais
http://sourceforge.net/projects/acbr/files/

Download direto da ferramenta ACBrInstall
http://www.projetoacbr.com.br/forum/index.php?/files/file/7-acbrinstall/

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Use Funções Nativas do Oracle Aplicadas no Delphi

Busca fonética e por similaridade

Resumo DevMan

Use Funções Nativas do Oracle Aplicadas no Delphi:

É muito comum utilizarmos o "like" quando precisamos realizar uma busca para resgatar dados, porém, o uso do mesmo nem sempre atende as necessidades, pois não levam em consideração possíveis erros de entrada de dados, ou seja, "erros humanos". Neste artigo abordaremos com exemplos práticos e teóricos como utilizar a função nativa do Oracle chamada JARO_WINKLER_SIMILARITY e saber mais detalhes sobre o pacote UTL_MATCH e suas funções, aplicando posteriormente estes conhecimentos no Delphi.

Em que situação o tema é útil:

Quando for necessário efetuar uma pesquisa textual, seja essa por um possível campo "Nome" ou qualquer outro que aceite texto livre, que requer um resultado não exato, ou seja, que deva retornar valores similares ao informado para a busca.

Hoje em dia o nível de complexidade das informações armazenadas no SGBD é cada vez maior, havendo assim uma necessidade crescente em encontrar novos meios de consulta. Um dos mais empregados atualmente, principalmente em projetos para web, são as buscas por similaridade. A necessidade em encontrar novos meios de consulta se deve à grande quantidade de inconsistências causadas por erros ortográficos, abreviações, caracteres trocados, siglas entre outros. Isto faz com que um mesmo objeto seja representado de várias formas, o que dificulta a busca deste pelos meios tradicionais.

Utilizando esta técnica de maneira apropriada é possível obter resultados satisfatórios, mesmo tendo pouco ou nenhum conhecimento do conteúdo armazenado. É comum a ocorrência de dados inconsistentes principalmente quando o mesmo é digitado ou quando o sistema é integrado, seja este representado como um endereço salvo com abreviações e erros ortográficos ou quando se quer resgatar dados do SGBD. Nestes casos um operador tradicional de busca não consegue recuperar os dados devido à variação existente, já que o mesmo utiliza a comparação por igualdade. Logo este problema não fica restrito somente a endereços, mas também aos mais variados casos.

Por exemplo, numa consulta para localizar um endereço cadastrado como "Avenida Presidente Vargas". O mesmo pode conter diversas variações como "Av. Presidente Vargas" ou "Av. Pres. Vargas". Note que todas as variações se referem ao mesmo elemento. Numa consulta tradicional se utilizássemos qualquer destas variações o endereço retornaria vazio.

Temos também vários exemplos na web de sites que utilizam o conceito de busca por similaridade. O Google é um exemplo. Basta reparar que quando é digitado algum termo, mesmo que errado, automaticamente é exibida uma sugestão para o termo, onde muitas vezes é o termo escrito de forma correta e corrigida. Além do Google temos vários outros exemplos na web, como buscas realizadas em sites de vendas para localizar algum produto. Podemos aplicar este conceito de busca no Delphi, tornando assim as consultas mais "inteligentes", servindo assim como um diferencial na aplicação e facilitador para o usuário. Nesse artigo utilizamos um recurso nativo do Oracle, o pacote UTL_MATCH.

Já com o operador "like", podemos comparar cadeias de caracteres utilizando padrões de comparação (SQL WildCard) para um ou mais caracteres, contudo, a pesquisa fica restrita a digitação correta.

Nota

WildCard é um operador coringa que utiliza padrões de comparação usados no SQL para realizar uma busca seletiva. Este é representado por caracteres como:"%" (substitui nenhum ou mais caracteres), "_" (substitui um caractere).

Esse pacote foi introduzido no Oracle 10g Release 2, mas como foi documentado posteriormente, é dito que foi oficialmente suportado somente no Oracle 11g Release 2. Contém uma variedade de funções que são úteis para testar o nível de semelhança/diferença entre cadeias de texto.

Abordamos um pouco sobre cada uma dessas funções, dando mais ênfase à função JARO_WINKLER_SIMILARITY e sua implementação no Delphi.

Introdução ao pacote UTL_MATCH

Neste pacote há quatro funções que utilizam métodos diferentes para comparar a cadeia de caracteres de origem e destino. Estes retornam uma avaliação do que seria necessário para transformar este determinado conteúdo de origem em destino. As funções são separadas em duas categorias, EDIT_DISTANCE e JARO_WINKLER, que possuem diferentes algoritmos de análise (**Nota do DevMan 1)**.

Nota do DevMan 1

Em 1965 um cientista russo chamado Vladimir Levenshtein, criou um algoritmo para calcular a distância ou semelhança entre duas cadeias de caracteres. O cálculo baseia-se na quantidade de operações de substituição, deleção e inserção necessárias para a cadeia de origem se transformar na cadeia de destino. O algoritmo é popularmente chamado de "distância de Levenshtein" em homenagem ao cientista russo criador do algoritmo ou também conhecido como "distância de edição" (EDIT_DISTANCE).

O algoritmo "JARO_WINKLER" é outra forma de calcular a semelhança entre duas cadeias de caracteres. Criado nos EUA para atender ao Censo, este estabelece uma métrica de distância entre duas cadeias, onde se calcula o número de caracteres comuns as duas strings e a semelhança da ordem em que se apresentam, sempre levando em conta possíveis erros humanos.

As quatro funções são dadas pelos seguintes nomes:
- EDIT_DISTANCE;
- EDIT_DISTANCE_SIMILARITY;
- JARO_WINKLER;
- JARO_WINKLER_SIMILARITY;

Função EDIT_DISTANCE

Função que retorna a quantidade de substituições, deleções e inserções necessárias para transformar a cadeia de origem em destino. Exemplo:

VICTOR: Cadeia de origem (a ser transformada);
VITOR: Cadeia de destino (valor procurado);

Um exemplo do modo prático de uso pode ser demonstrado na função da **Listagem 1**.

Listagem 1. Exemplo de uso da função EDIT_DISTANCE

```
01 SELECT UTL_MATCH.EDIT_DISTANCE('VICTOR','VITOR') FROM DUAL;
```

Ao executar a função exibida na **Listagem 1**, o resultado será equivalente ao valor 1.

Função EDIT_DISTANCE_SIMILARITY

A função EDIT_DISTANCE_SIMILIRITY é responsável por retornar a quantidade de substituições, deleções e inserções necessárias para transformar a cadeia de origem em destino. Sua execução retorna um percentual de similaridade de zero a cem por cento. A **Listagem 2** ilustra o modo de utilização da função.

O retorno da **Listagem 2** é o valor de 84 % (índice de similaridade).

Listagem 2. Utilização da função EDIT_DISTANCE_SIMILARITY

```
01 SELECT UTL_MATCH.EDIT_DISTANCE_SIMILARITY('VICTOR','VITOR') FROM DUAL;
```

Função JARO_WINKLER

Tem por finalidade calcular a medida de similaridade entre as cadeias de origem e destino, sempre levando em consideração possíveis entradas de erros. A **Listagem 3** ilustra seu modo de uso.

Listagem 3. Uso da função JARO_WINKLER

```
01 SELECT UTL_MATCH.JARO_WINKLER('VICTOR','VITOR') FROM DUAL;
```

O retorno da similaridade obtido na **Listagem 3** é uma dízima periódica equivalente ao valor 0,9555555555555555.

Função JARO_WINKLER_SIMILARITY

Função que calcula a medida de similaridade entre a cadeia de origem e destino sempre levando em consideração possíveis entradas de erros. Retorna um percentual de similaridade de zero a cem por cento. A **Listagem 4** ilustra um exemplo de uso desta função.

O retorno obtido da execução da SQL exibida na **Listagem 4** é o valor de 95% de similaridade entre a cadeia de origem ('VICTOR') e a cadeia de destino ('VITOR');

Listagem 4. Exemplo de uso da função JARO_WINKLER_SIMILARITY

```
01 SELECT UTL_MATCH.JARO_WINKLER_SIMILARITY('VICTOR','VITOR') FROM DUAL;
```

Instalação do Oracle XE

Os recursos apresentados também estão disponíveis nas versões Express do Oracle. Ao acessar o site de download (seção Links) podemos realizar o download da versão 11g Release 2 XE. A versão XE é gratuita para os administradores de banco de dados, desenvolvedores, estudantes e instituições educacionais. Esse produto da Oracle é totalmente gratuito para baixar, desenvolver e distribuir. Possui algumas limitações, mas irá permitir que os desenvolvedores criem soluções de ponta que poderão ser portadas para as versões pagas do produto sem grandes problemas. Dentre as limitações do Oracle Database XE estão:
- Tamanho do banco até 11 GB;
- Somente um banco de dados por máquina, ou seja, um servidor poderá ter apenas um banco Oracle;
- Limite de uso de 1 GB de RAM;
- Uso de 1 processador somente, mesmo em ambiente com mais de um processador.

O arquivo de instalação é distribuído de forma compactada, em um zip, que possui aproximadamente 312MB. Salve o arquivo OracleXE112_Win32.zip em uma pasta de sua preferência. Quando o download for finalizado, descompacte. Existirá uma pasta Disk1, ao abri-la temos o arquivo setup.exe, execute-o para iniciar a instalação. A instalação é bastante intuitiva, contudo, é preciso atenção, pois durante o processo é pedida uma senha para a conta de sistema do gerenciador do banco de dados.

Criação de um banco de dados

Uma vez concluída a instalação é possível acessar o Oracle APEX (**Figura 1**) através do navegador de sua preferência, no endereço: http://127.0.0.1:8080/apex/f?p=4950. O Oracle Application Express (APEX) é uma ferramenta de desenvolvimento que pode ser utilizada via navegador e pode ser utilizada com qualquer versão do Oracle.

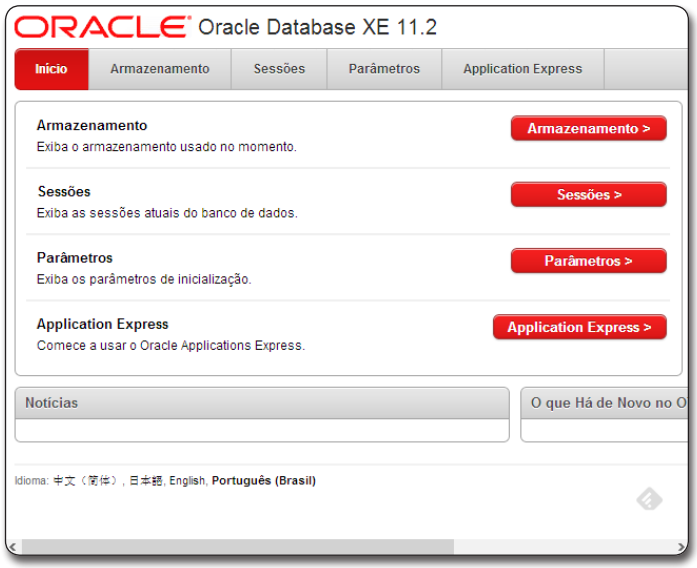


Figura 1. Oracle APEX

A criação de um banco de dados pode ser acessada através do botão Application Express. Ao escolher essa opção, será solicitado o usuário (SYSTEM) e a senha cadastrada durante a instalação. Então será exibida a tela de configuração de um novo banco (**Figura 2**).

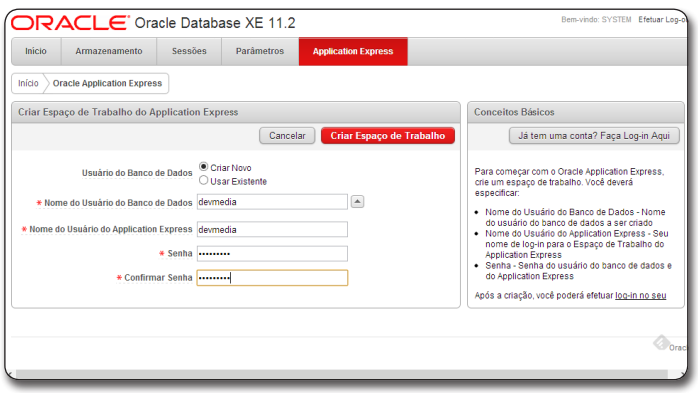


Figura 2. Criação de um novo banco de dados

O Oracle trabalha de uma maneira diferente em relação a alguns bancos de dados, principalmente em relação ao Firebird, que é o grande conhecido dos desenvolvedores Delphi. No Oracle existe o conceito de WorkSpace ou Espaço de Trabalho. Ao criar um novo espaço de trabalho, um novo banco de dados é criado.

Após informar o nome de usuário e uma senha para o banco de dados a ser criado, escolhemos o botão Criar Espaço de Trabalho. Nesse momento será criado o Espaço de Trabalho. Ao término, podemos então realizar o login utilizando o nome de login e senha criados. Ao realizar esse login o APEX nos abre uma tela de administração, onde é possível criar novas tabelas, aplicações e muito mais (**Figura 3**).

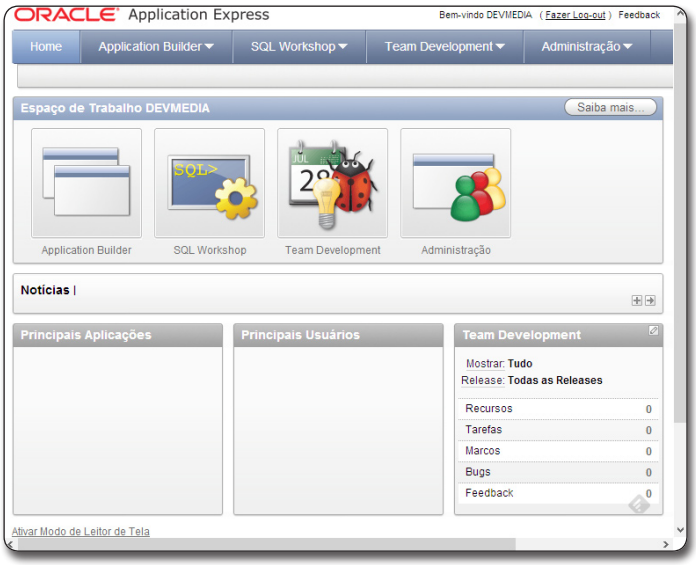


Figura 3. Espaço de Trabalho DEVMEDIA

Definição das tabelas

Criaremos uma tabela no Oracle denominada de CLIENTE. A instrução utilizada para a criação desta tabela encontra-se na **Listagem 5**. Para executá-la, acessamos a opção SQL WorkShop do APEX e então escolhemos Scripts SQL. Na tela que abre temos um botão em destaque, chamado Criar. Ao clicar nesse botão podemos digitar o script para então clicar no botão Executar, ou podemos também na opção SQL WorkShop acessar SQL Commands, digitar o script e executa- ló pelo clique do botão Executar.

Listagem 5. Script da tabela CLIENTE

```
01 CREATE TABLE CLIENTE (
02  IDCLIENTE          NUMBER,
03  DESCRICAO          VARCHAR2(255 BYTE),
04  ENDERECO           VARCHAR2(255 BYTE),
05  MUNICIPIO          VARCHAR2(255 BYTE),
06  UF                 VARCHAR2(255 BYTE),
07  REPRESENTANTE      VARCHAR2(255 BYTE),
08  TELEFONE           VARCHAR2(255 BYTE),
09  CEP                VARCHAR2(255 BYTE),
10  CONSTRAINT IDCLIENTE_PK PRIMARY KEY (IDCLIENTE));
```

Como se pode notar, a **Listagem 5** ilustra uma tabela simples, apenas com alguns poucos campos que permitam realizar exemplos com as funções aqui apresentadas. Na **Listagem 6** são realizadas algumas instruções de INSERT para fins de teste. Podemos executar esse script da mesma forma que o primeiro foi.

Listagem 6. Inserção de valores na tabela CLIENTE

```
01 INSERT INTO CLIENTE VALUES(1,'MATHEUS LYRA','RUA DO CARMO',
  'RIO DE JANEIRO','RJ','PAULO AFONSO','(21)3333-3333','22777-111');

02 INSERT INTO CLIENTE VALUES(2,'RODRIGO MOURAS','RUA SETE DE SETEMBRO',
  'RIO DE JANEIRO','RJ','PAULO AFONSO','(21)4444-4444','33333-222');

03 INSERT INTO CLIENTE VALUES(3,'THIAGO MOURA','RUA TRAVESSA DO
  OUVIDOR','RIO DE JANEIRO','RJ','SERGIO MOURA','(21)5555-5555',
  '55555-111');

04 INSERT INTO CLIENTE VALUES(4,'RODOLFO SILVA','RUA DA QUITANDA',
  'RIO DE JANEIRO','RJ','SERGIO MOURA','(21)6666-6666','66666-111');

05 INSERT INTO CLIENTE VALUES(5,'MATEUS SIMAS','RUA DO OUVIDOR',
  'RIO DE JANEIRO','RJ','PAULO AFONSO','(21)7777-3333','88888-111');
```

Podemos conferir o resultado efetuando uma sentença de busca. Basta acessar a opção SQL Workshop, então escolher Comandos SQL e executar a sentença SELECT * FROM CLIENTE. Essa realizar uma busca completa na tabela Cliente, como mostra a **Figura 4**.

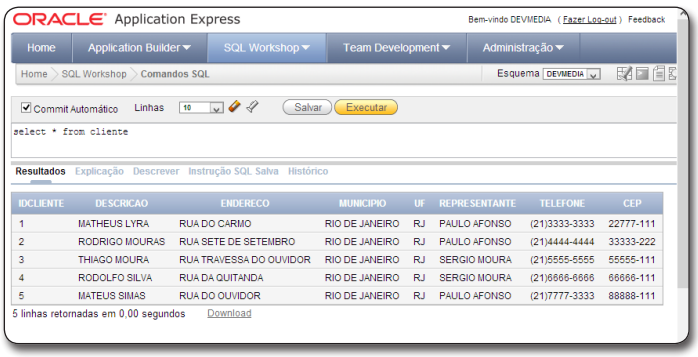


Figura 4. Tabela cliente preenchida

Instalação do Client

Para instalar o client do Oracle acesse o site disponível na seção **Links** e baixe o arquivo "win32_11gR2_client.zip". Durante a instalação escolha o tipo de instalação "Administrador". Quando for especificar a localização da instalação, faça de acordo com a **Figura 5.**

Após concluir a instalação é necessário configurar o Client do Oracle e o Odbc. Para configurar o Client do Oracle acesse: Oracle - OraClient11g_home1 > Ferramentas de Configuração e Migração > Assistente de Administração para Windows. Escolhendo a opção: Configuração do Nome do serviço de rede local, definimos o acesso ao Banco de Dados criado ou outro serviço através da rede, então vamos adicionar um acesso. Clique em Próximo para então configuramos um nome de serviço de Banco de Dados que é normalmente o Banco de Dados global, no nosso caso o nome do serviço é "XE" que é o serviço padrão do Oracle.

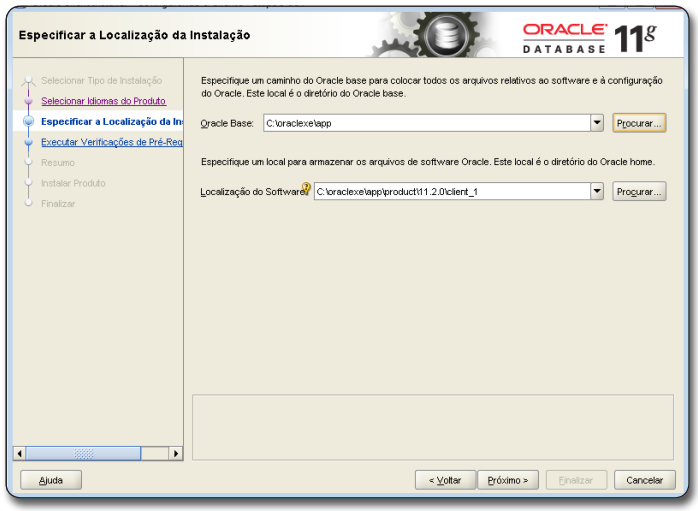


Figura 5. Instalação do Client

Na próxima tela defina o protocolo de comunicação do banco de dados na rede como TCP e clique em próximo. Logo após temos a opção para estabelecer comunicação com o banco de dados usando o protocolo TCP/IP, mas é necessário o nome do host do computador que contém o banco de dados. Já que instalamos localmente o Banco de dados definimos nome do host como localhost. Utilize o número de porta padrão 1521. O passo seguinte é tentar estabelecer conexão com o banco de dados a partir das configurações definidas anteriormente, contudo é necessário alterar o login e senha antes disso. Utilize o login e senha de acesso ao BD. Terminado o teste e estabelecida a conexão, configuramos o ODBC: Oracle - OraClient11g_home1 > Ferramentas de Configuração e Migração > Administrador do Microsoft ODBC. O ODBC é um middleware, que define um conjunto de interfaces e permite o uso de linguagens de programação variadas, esta tecnologia faz a intermediação entre o Banco de Dados e a aplicação. Em Fonte de Dados do Sistema adicionamos uma nova fonte utilizando o Drive: Oracle OraClient 11g_home1. Configure o TNS Service Name para XE que é o serviço de rede local que configuramos anteriormente. Defina o DataSouce Name, User ID como DEVMEDIA.

A aplicação

Após isso, iniciamos uma aplicação do tipo VCL Application selecionando no menu a opção New>VCL Application – Delphi. Renomeamos o formulário para FrmBuscaSimilaridade e adicionamos os componentes a seguir de forma que o formulário fique com aparência semelhante à da **Figura 6**.

- Edit,
- Button,
- DBGrid,
- SQLConection,
- SQLQuery

- DataSetProvider
- ClientDataSet
- DataSource.

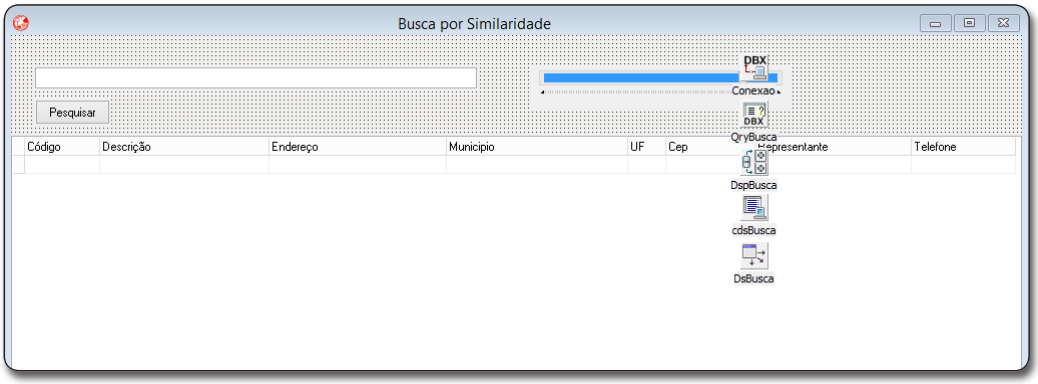


Figura 6. Tela pesquisa de clientes

Vamos configurar as propriedades do componente SQLConnection (nomeado para Conexao) de acordo com o tipo de banco de dados utilizado. Defina a propriedade Drive como Oracle, ConnectionName com qualquer nome pois se trata de uma alias que identifica a conexão estabelecida, em params defina o DataBase como XE, UserName e Password com o dados configurados de acesso ao BD. Este componente representa diretamente o elo entre o banco de dados e a aplicação, mas não realiza acesso direto ao conteúdo das tabelas. No componente SQLQuery renomeie-o para qryBusca, configurando a propriedade SQLConnection com o componente Conexao. O SQLQuery é o componente que se encarrega de executar as consultas SQL e é unidirecional.

O DataSetProvider é o provedor de acesso ao banco de dados. Sua propriedade Name deve ser setada para dspBusca e a propriedade DataSet apontando qryBusca.

O ClientDataSet (cdsBusca) trabalha com o conteúdo desconectado, em forma de cache, o que torna o manuseio dos dados muito mais performático. Aponte sua propriedade ProviderName para o DataSetProvider dspBusca.

O DataSource é responsável por realizar a ligação entre a fonte de dados (ClientDataSet neste caso) e os componentes visuais. Como uma última configuração, basta renomeá-lo para dsBusca e apontar sua propriedade DataSet para o cdsBusca, assim a ligação já está realizada. Agora que temos os principais componentes de conexão e consulta previamente configurados, partimos para os componentes visuais. O Edit foi renomeado para edtBusca e o Button para btnPesquisar. Note que sempre que nomear um componente, utiliza-se um padrão de nomenclatura apenas por convenção de boas práticas, já que isso facilita na hora de localizar e utilizar algum deles no código. Conecte o DataSource ao DBGrid através da propriedade de mesmo nome, setando o componente dsBusca. Finalizada a parte visual, implementaremos a instrução SQL responsável por fazer a busca fonética na nossa tabela anteriormente definida. Sendo assim, adicione a instrução SQL da **Listagem 7** à propriedade SQL do componente qryBusca.

O script seleciona todos os campos da tabela CLIENTE e traz os registros de acordo com o grau de similaridade. Como filtro, é utilizado o retorno da função **JARO_WINKLER_SIMILARITY** que retorna um número inteiro entre zero e cem, conforme já mencionado. Neste caso, considera-se que o valor deve ser maior que 85% de similaridade. Observe ainda na mesma listagem a passagem dos parâmetros referentes às cadeias de origem (nome do cliente) e destino (campo da tabela) necessários para função JARO_WINKLER_SIMILARITY.

Após definida a instrução de consulta, é criada uma procedure no Delphi denominada de Pesquisa. O código é ilustrado na **Listagem 8.**

Listagem 7. Instrução SQL para consulta

```
01 SELECT
02  IDCLIENTE,
03  DESCRICAO,
04  ENDERECO,
05  MUNICIPIO,
06  UF,
07  REPRESENTANTE,
08  TELEFONE, CEP
09 FROM
10  CLIENTE
11 WHERE
12  UTL_MATCH.JARO_WINKLER_SIMILARITY(UPPER(:NOMECLIENTE),
    DESCRICAO) > 85;
```

Listagem 8. Procedure Pesquisa

```
01 procedure TFrmBuscaSimilaridade.Pesquisa(pDescricao: String);
02 begin
03  with cdsBusca do
04  begin
05   Close;
06   Params.ParamByName('NOMECLIENTE').AsString := pDescricao;
07   Open;
08  end;
09 end;
```

A procedure recebe o parâmetro de entrada pDescricao (linha 01) que é repassado ao parâmetro NOMECLIENTE do ClientDataSet cdsBusca (linhas 03 a 06). Observe que antes da consulta, o DataSet é fechado (linha 05) e em seguida aberto (linha 08).

Para fazer uso da procedure Pesquisa utilizamos os eventos onClick do botão de pesquisa e o evento OnChange do Edit. Observe a implementação idêntica nas **Listagens 9** e **10**, respectivamente.

Como pode ser observado, o método Pesquisa é chamado passando como parâmetro o conteúdo digitado no componente TEdit, através da propriedade Text. A diferença de comportamento entre as duas se dará que, na primeira a pesquisa ocorrerá apenas no ato de clicar no botão, quando na segunda ocorrerá a cada caractere digitado no TEdit.

Listagem 9. Chamada da pesquisa no evento OnClick do botão

```
01 procedure TFrmBuscaSimilaridade.btnPesquisarClick(Sender: TObject);
02 begin
03   Pesquisa(edtBusca.Text);
04 end;
```

Listagem 10. Chamada da pesquisa no evento OnChange do Edit

```
01  procedure TFrmBuscaSimilaridade.edtBuscaChange(Sender: TObject);
02 begin
03   Pesquisa(edtBusca.Text);
04 end;
```

Com o código feito até aqui já temos uma consulta em funcionamento, porém, apenas para agregar um efeito visual à nossa aplicação vamos adicionar um componente TrackBar. Este será o responsável por exibir o grau de similaridade da consulta realizada. Adicione-o configurando seu Name para TbSimilaridade e as respectivas propriedades Min para 0, Max para 100 e Position com o valor 50.

Modificando o comportamento do nosso projeto, a Query e o método Pesquisa serão alterados para trabalharem com o parâmetro de similaridade. Sendo assim, altere o componente SQLQuery informando a instrução da **Listagem 11.**

Listagem 11. Consulta alterada no componente SQLQuery

```
01 SELECT
02   IDCLIENTE,
03   DESCRICAO,
04   ENDERECO,
05   MUNICIPIO,
06   UF,
07   REPRESENTANTE,
08   TELEFONE,
09   CEP
10 FROM
11   CLIENTE
12 WHERE
13   UTL_MATCH.JARO_WINKLER_SIMILARITY(UPPER(:NOMECLIENTE),
     DESCRICAO)  > :POSICAO;
```

Observe que a única parte que teve uma mudança significativa foi a condição where (linha 13) que foi modificada do valor fixo de 85 para o parâmetro **posicao**, que será preenchido a seguir.

No script da **Listagem 12** o método Pesquisa realiza o mesmo procedimento anterior, porém, agora passando o valor da propriedade Position do TrackBar para a o parâmetro informado na instrução SQL modificada (linha 07). Note que isto fará com que o filtro retorne apenas os registros que possuem esse índice de similaridade.

Nota

Lembre-se de sempre que modificar os parâmetros de um DataSet, executar a instrução Fetch Params do ClientDataSet, clicando de direita sobre o mesmo.

Executando o programa será possível observar a pesquisa de acordo com o TrackBar. Como mencionado anteriormente, é possível utilizar estes métodos no Oracle por meio de uma função. Sendo assim, a **Listagem 13** ilustra a função F_SIMILARIDADE que realiza o mesmo procedimento da função **EDIT_DISTANCE_SIMILARITY** no Oracle. Esta calcula o percentual de similaridade a partir do algoritmo de Vladimir Levenshtein conhecido como "Distância Levenshtein".

Listagem 12. Procedure Pesquisa Alterada

```
01 procedure TFrmBuscaSimilaridade.Pesquisa(pDescricao: String);
02 begin
03   with cdsBusca do
04   begin
05     Close;
06     Params.ParamByName('NOMECLIENTE').AsString := pDescricao;
07     Params.ParamByName('POSICAO').AsInteger   := TbSimilaridade.Position;
08     Open;
09   end;
10 end;
```

Listagem 13. Função Oracle que realiza busca por similaridade

```
01 CREATE OR REPLACE
02 FUNCTION F_SIMILARIDADE
03  (P_SOURCE_STRING   IN VARCHAR2,
04   P_TARGET_STRING   IN VARCHAR2)
05  RETURN             NUMBER
06  DETERMINISTIC
07 AS
08   V_LENGTH_OF_SOURCE    NUMBER := NVL (LENGTH (TRIM(P_SOURCE_STRING)), 0);
09   V_LENGTH_OF_TARGET    NUMBER := NVL (LENGTH (TRIM(P_TARGET_STRING)), 0);
10   TYPE MYTABTYPE IS     TABLE OF NUMBER INDEX BY BINARY_INTEGER;
11   COLUMN_TO_LEFT        MYTABTYPE;
12   CURRENT_COLUMN        MYTABTYPE;
13   V_COST                NUMBER := 0;
14 BEGIN
15   IF V_LENGTH_OF_SOURCE = 0 OR V_LENGTH_OF_TARGET = 0 THEN
16     RETURN 0;
17   ELSE
18     FOR J IN 0 .. V_LENGTH_OF_TARGET LOOP
19       COLUMN_TO_LEFT(J) := J;
20     END LOOP;
21     FOR I IN 1.. V_LENGTH_OF_SOURCE LOOP
22       CURRENT_COLUMN(0) := I;
23       FOR J IN 1 .. V_LENGTH_OF_TARGET LOOP
24         IF SUBSTR (UPPER(TRIM(P_SOURCE_STRING)), I, 1) =
25           SUBSTR (UPPER(TRIM(P_TARGET_STRING)), J, 1)
26         THEN V_COST := 0;
27         ELSE V_COST := 1;
28         END IF;
29         CURRENT_COLUMN(J) := LEAST (CURRENT_COLUMN(J-1) + 1,
30                                     COLUMN_TO_LEFT(J) + 1,
31                                     COLUMN_TO_LEFT(J-1) + V_COST);
32       END LOOP;
33       FOR J IN 0 .. V_LENGTH_OF_TARGET  LOOP
34         COLUMN_TO_LEFT(J) := CURRENT_COLUMN(J);
35       END LOOP;
36     END LOOP;
37   END IF;
38   RETURN 100*(1-(CURRENT_COLUMN(V_LENGTH_OF_TARGET)
       /V_LENGTH_OF_SOURCE));
39 END F_SIMILARIDADE;
```

Na função F_SIMILARIDADE temos dois parâmetros de entrada P_SOURCE_STRING e P_TARGET_STRING (linhas 03 e 04) que representam a cadeia de origem e destino respectivamente.

O retorno da função (linhas 05 e 06) é do tipo Number, que receberá valores que irão variar entre zero e cem por cento dependendo do grau de similaridade. A seguir, as variáveis V_LENGTH_OF_SOURCE e V_LENGTH_OF_TARGET (linhas 08 e 09) são do tipo Number e receberão como valores a quantidade de caracteres das respectivas cadeias. As variáveis COLUMN_TO_LEFT e CURRENT_COLUMN (linhas 11 e 12) são nossas matrizes indexadas que receberão os valores calculados até chegar ao resultado final. A variável V_COST (linha 13) receberá o valor do custo. Na linha 15 temos uma validação que verifica se foram ou não passados os valores por parâmetro, retornando 0 e encerrando a função caso a validação seja verdadeira. Entre as linhas 18 e 20, preenchemos a primeira matriz passando os valores de zero até o tamanho de caracteres da cadeia de destino. Após este processo, realizamos loops para preenchimento da matriz responsável pela cadeia origem com os valores calculados de acordo com as validações apresentadas. A primeira verificação (linhas 24 a 28) valida se o caractere da matriz origem é igual ao caractere da matriz destino, então a variável V_COST receberá o valor 0, caso contrário, o valor 1. A parte principal do cálculo do algoritmo que chamamos de função mínima (linhas 29 a 31) é composta por três valores responsáveis por quantificar a deleção, inserção e substituição respectivamente, onde destes três valores, somente o mínimo será adicionado à matriz CURRENT_COLUMN. Após a conclusão do processo, o resultado se encontra na última posição da matriz localizada no canto inferior direito (linha 38). A **Tabela 1** faz uma representação gráfica do algoritmo.

		V	I	T	O	R
	0	1	2	3	4	5
V	1	0	1	2	3	4
I	2	1	0	1	2	3
C	3	2	1	1	2	3
T	4	3	2	1	2	3
O	5	4	3	2	1	2
R	6	5	4	3	2	1

Tabela 1. Resultado algoritmo de Levenshtein

Listagem 14. Fórmula aplicada no algoritmo

```
01 Matriz[i,j]= Mínimo(
02     Matriz[i-1, j]+1,
03     Matriz[i, j-1]+1,
04     Matriz[i-1,j-1]+custo)
```

De acordo com a **Tabela 1** a última linha e a última coluna da matriz representam o resultado final (1), ou seja, para transformar a cadeia de origem (VICTOR) na cadeia destino (VITOR) é preciso apenas uma exclusão. Note que os zeros identificam os caracteres iguais entre as duas cadeias. A fórmula aplicada para isto é demonstrada na **Listagem 14**.

Funciona da seguinte maneira: A posição i é correspondente a linha de nossa matriz e j refere-se a coluna. O código na linha 02 representa o quantificador da operação de deleção, linha 03 a operação inserção e linha 04 a operação de substituição. O valor mínimo entre estas três operações será adicionado à matriz principal. Lembre-se que os caracteres iguais, o valor do custo é 0 senão é 1. Na **Listagem 15** temos dois exemplos práticos com base na **Tabela 1**.

No primeiro exemplo da **Listagem 15** (linha 01 a 04) temos o valor 0 (linha02) que corresponde a linha da matriz e o valor 1 que representa a coluna, basta então verificarmos em nossa matriz principal qual é o valor referente a esta posição, que neste caso, equivale a 1. Depois somamos um, cujo resultado é dois. Basta seguir o mesmo conceito para os demais cálculos. Repare que na linha 04 do nosso exemplo, temos o custo como zero. Isto porque nesta comparação há caracteres iguais. Dentre estes três valores, apenas o mínimo será adicionado à matriz principal no momento do cálculo. No primeiro exemplo, a matriz principal recebe zero na posição "linha um e coluna um". No exemplo seguinte, (linhas 05 a 08) segue a sequência, incrementando a posição das matrizes até o fim da matriz principal.

Similar ao que é efeito no Oracle, a **Listagem 16** exibe uma função em Delphi que calcula a Distância Levenshtein.

Listagem 15. Exemplo prático

```
01 Matriz[1,1] = Mínimo(
02     Matriz[0,1]+1 = 2
03     Matriz[1,0]+1 = 2
04     Matriz[0,0]+0 = 0 )

Próximo
05 Matriz[1,2] = Mínimo(
06     Matriz[0,2]+1 = 3
07     Matriz[1,1]+1 = 1
08     Matriz[0,1]+0 = 2)
```

Listagem 16. Função Delphi que realiza busca por similaridade

```
01 function Levenshtein(Word1, Word2: String): integer;
02 var lev : array of array of integer;
03    i,j : integer;
04    s : string;
05 begin
06   result := 0;
07  if (Length(Word1) = 0) or (Length(Word2) = 0) then
08    exit;
09  SetLength(lev, length(Word1) + 1);
10  for i := low(lev) to high(lev) do
11    setLength(lev[i], length(Word2) + 1);
12  for i := low(lev) to high(lev) do lev[i][0] := i;
13  for j := low(lev[low(lev)]) to high(lev[low(lev)]) do lev[0][j] := j;
14  for i := low(lev)+1 to high(lev) do
15  for j := low(lev[i])+1 to high(lev[i]) do
      lev[i][j] := min(min(lev[i-1][j] + 1, lev[i][j-1] + 1) ,lev[i-1][j-1] + ifthen(Word1[i] =
      Word2[j], 0, 1));
16  result := trunc(100*(1- ((lev[length(word1)][length(word2)])/length(word1))));
17 end;
```

Seguimos o mesmo conceito visto no Oracle. Aplicamos a fórmula apresentada na **Listagem 13** para calcular a similaridade entre duas cadeias de caracteres. Nesta função utilizamos um array multidimensional (linha 02). Preenchemos as posições referentes a linha e coluna da matriz com valores que vão de

zero até as quantidades de caracteres de cada cadeia de origem e destino (linhas 12 e 13). Após o processo, calculamos o mínimo (linha 15) e adicionamos à matriz na posição determinada na hora da sequência. Como retorno, temos o percentual de similaridade (linha 16).

Conclusão

Neste artigo, conhecemos mais sobre o pacote UTL_MATCH e suas funções que fazem parte deste grande universo de recursos do Oracle. Aprendemos na teoria e na prática como realizar uma busca por similaridade de forma simples e prática.

Como visto e demonstrado, este é um recurso que aperfeiçoa a maneira como realizamos uma pesquisa, afinal de contas, qual sistema não utiliza uma tela de pesquisa? Qual site de vendas não precisa de uma busca por similaridade? Qual o usuário está isento de erros de digitação? Tudo depende da necessidade de cada um, do que cada um julga necessário ou não. Foi apresentado o recurso como uma forma diferencial de aprimorar suas soluções, permitindo ainda assim ser expandido para consultas mais elaboradas utilizando os operadores apresentados.

Matheus de Lima Lyra
mdlimalyra@gmail.com
Graduado em Análise e Desenvolvimento de sistemas, com mais de 3 anos de experiência como programador Delphi, desenvolvendo e realizando manutenção em diversos sistemas. Experiência em PL/SQL utilizando Banco de Dados Oracle, MySql e SqlServer.

Download do Oracle 11g Release 2 Express
http://www.oracle.com/technetwork/products/express-edition/downloads/index.html

Download do Client Oracle 11g Release 2
http://www.oracle.com/technetwork/database/enterprise-edition/downloads/112010-win32soft-098987.html

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior portal para desenvolvedores da América Latina!

20 mil posts

430 mil cadastrados

10 milhões de page-views por mês

http://www.devmedia.com.br/

DevMedia

Mapeamento ORM com o TMS Aurelius

Diferentes cenários de mapeamento

A programação orientada a objetos está cada vez mais popular, e isso não é diferente com o programadores Delphi. Este fato se deve a gama de vantagens que essa tecnologia apresenta, entre elas, podemos destacar o desenvolvimento de aplicações mais otimizadas, códigos melhores organizados, e abstração da interação com o banco de dados, sem mencionar um reaproveitamento de código através de reuso de classes, aplicação de herança e outras características da orientação a objetos. Essas e outras vantagens permitem o programador ter mais tranquilidade nas futuras manutenções de seus softwares. Mas o uso de orientação a objetos resultou em um pequeno problema. É indiscutível a popularidade do uso de bancos de dados relacionais para armazenamento de dados. Eles possuem seu mérito, estão no mercado a há anos e já se mostraram mais que eficientes. Porém, os conceitos relacionais são limitados quando comparados aos conceitos orientados a objetos. Isso gera uma diferença conhecida como impedância objeto-relacional. Buscando a solução para esse "vazio" entre as duas tecnologias, foi criada uma ponte, um mecanismo que ligue as duas realidades. Assim, nasceram os frameworks ORM, que realizam o mapeamento dos dados de lado para o outro. De uma forma bem simples, um ORM nada mais é do que um conjunto de classes que permitem dados expressos em classes serem salvos em bancos relacionais, sem que se faça necessário escrever códigos de conexão com o banco ou consultas SQL a todo o momento, dessa forma se preservam as características da orientação a objetos. Com um ORM é possível mapear uma entidade do mundo real para o ambiente computacional e fazer com que as regras de negócios sejam expressas em classes, e posteriormente salvas em um banco relacional.

Vários programadores Delphi estão aderindo às novas tendências do mercado e com o passar do tempo o RAD recebeu vários novos recursos para trabalhar com essas tecnologias, podemos citar alguns frameworks como o DORM e o Aurelius que nos permitem trabalhar da mesma forma dos outros ambientes de desenvolvimento, neste artigo nós vamos trabalhar com o framework ORM Aurelius.

Resumo DevMan

Mapeamento ORM com o TMS Aurelius:

Neste artigo vamos conferir o funcionamento do framework ORM da TMS para Delphi, o Aurelius. Serão abordados alguns conceitos sobre ORM, além de particularidades do TMS Aurelius, como por exemplo, os SQL Dialects. Neste artigo também iremos conferir como fazer a conexão com um banco de dados e ver que é possível trabalhar com bancos de dados já existentes ou criar novos bancos baseados na estrutura de classes de sua aplicação. Isso é possível através dos mapeamentos das classes.

Em que situação o tema é útil:

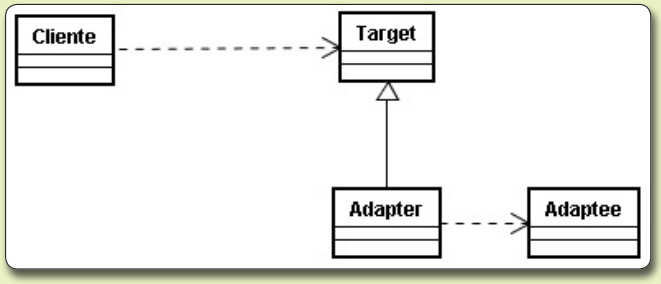
Quando se deseja desenvolver utilizando a orientação a objetos mas, o sistema de armazenamento de dados é um banco relacional. Para realizar a transformação dos dados para objetos e vice versa é necessário aplicar um ORM para diminuir as diferenças entre tabelas e classes.

O Aurelius

O Aurelius foi criado pela TMS, apesar de novo no mercado, é um completo framework que permite criar aplicações com todas as vantagens citadas sobre OO e ORM e possui um manual completo que mostra como a ferramenta funciona (ver seção **Links**). O Aurelius nos permite trabalhar com vários bancos de dados, existe um adaptador para cada componente de acesso no Delphi e para utilizar é muito simples, basta fazer o uso da Unit no seu código, por exemplo, para utilizar o dbExpress o Aurelius disponibiliza o adaptador TDBExpressConnectionAdapter que pode ser utilizado através da Unit Aurelius.Drivers.dbExpress. Todos os adaptadores estão declarados na Unit Aurelius.Drivers.XXXXX, onde XXXXX é o nome da tecnologia de acesso a dados. A **Tabela 1** mostra todos os adaptadores suportados. Um adaptador segue o conceito do padrão de projeto adapter (**Nota do DevMan 1**).

Nota do DevMan 1

O padrão adapter faz com que uma aplicação conheça uma interface única para acessar determinadas classes. Este pattern é interessante para evitar problemas como troca de versão de componentes de terceiros e também troca do componente em si por outro. Observe na figura ao lado onde temos a representação do padrão. Temos a interface Target que é a classe que o cliente irá conhecer, Adapter é a classe que fará o redirecionamento ao Adaptee que é o objeto adaptado e que possui a implementação real do código. Caso haja alguma alteração nos métodos de Adaptee, só teremos que mexer no nosso Adapter, e caso queiramos trocar o Adapter para usar outro framework, por exemplo, só precisaremos mexer no nosso Adapter, assim abstraímos mais a nossa aplicação e a tornamos mais fácil de ser alterada.



Absolute Database	
Classe do Adaptador	TAbsoluteDBConnectionAdapter
Unit	Aurelius.Drivers.AbsoluteDB
Componente	TABSDatabase
Fabricante	http://www.componentace.com
AnyDac	
Classe do Adaptador	TAnyDacConnection
Unit	Aurelius.Drivers.AnyDac
Componente	TADDatabase
Fabricante	http://www.dasoft.com/anydac
dbExpress	
Classe do Adaptador	TDBExpressConnectionAdapter
Unit	Aurelius.Drivers.dbExpress
Componente	TSQLConnection
Fabricante	Nativo Delphi
dbGo (ADO)	
Classe do Adaptador	TDbGoConnectionAdapter
Unit	Aurelius.Drivers.dbGo
Componente	TADOConnection
Fabricante	Nativo Delphi
Direct Oracle Access (DOA)	
Classe do Adaptador	TDoaConnectionAdapter
Unit	Aurelius.Drivers.Doa
Componente	TOracleSession
Fabricante	http://www.allroundautomations.com
ElevateDB	
Classe do Adaptador	TElevateDBConnectionAdapter
Unit	Aurelius.Drivers.ElevateDB
Componente	TEDBDatabase
Fabricante	http://elevatesoftware.com/
FIBPlus	
Classe do Adaptador	TFIBPlusConnectionAdapter
Unit	Aurelius.Drivers.FIBPlus
Componente	TFIBDatabase
Fabricante	http://www.devrace.com/

IBObjects (IBO)	
Classe do Adaptador	TIBObjectsConnectionAdapter
Unit	Aurelius.Drivers.IBObjects
Componente	TIBODatabase
Fabricante	http://www.ibobjects.com
Interbase Express (IBX)	
Classe do Adaptador	TIBExpressConnectionAdapter
Unit	Aurelius.Drivers.IBExpress
Componente	TIBDatabase
Fabricante	Nativo Delphi
NexusDB	
Classe do Adaptador	TNexusDBConnectionAdapter
Unit	Aurelius.Drivers.NexusDB
Componente	TnxDatabase
Fabricante	http://www.nexusdb.com
SQL-Direct	
Classe do Adaptador	TSQLDirectConnectionAdapter
Unit	Aurelius.Drivers.SqlDirect
Componente	TSDDatabase
Fabricante	http://www.sqldirectsoft.com
SQLite	
Classe do Adaptador	TSQLiteNativeConnectionAdapter
Unit	Aurelius.Drivers.SQLite
Componente	(Não Aplicável)
Fabricante	Nativo TML Aurelius
UniDac	
Classe do Adaptador	TUniDacConnectionAdapter
Unit	Aurelius.Drivers.UniDac
Componente	TUniConnection
Fabricante	http://www.devart.com/unidac
Unified Interbase (UIB)	
Classe do Adaptador	TUIBConnectionAdapter
Unit	Aurelius.Drivers.UIB
Componente	TUIBDatabase
Fabricante	http://sourceforge.net/projects/uib/

Tabela 1. Adaptadores suportados pelo Aurelius

Para salvar e manipular objetos no banco de dados, o Aurelius constrói e executa comandos SQL. Esses comandos são automaticamente ajustados através de objetos chamados de dialetos. Dessa forma o Aurelius consegue gerar sentenças SQL específicas para o gerenciador de banco de dados utilizado. Cada dialeto é representado por uma Unit que deve ser adicionada ao projeto. Essas Units possuem o seguinte padrão de nomenclatura: Aurelius.SQL.XXXXX, onde XXXXX é o nome do dialeto. A **Tabela 2** exibe os dialetos suportados pelo Aurelius.

Entendendo o acesso a um banco de dados

O Aurelius precisa do objeto de conexão para conectar-se a um banco de dados. Uma vez selecionado qual o tipo de banco de dados, é necessário então configurar adequadamente qual o dialeto a ser utilizado. O código da **Listagem 1** a seguir mostra como podemos realizar uma conexão utilizando o dbExpress.

Listagem 1. Instanciando o Adaptador

```
MinhaConexao := TDBExpressConnectionAdapter.Create(SQLConnection1, False);
```

O Aurelius também disponibiliza adaptador nativo para SQLite, você apenas precisar ter a DLL sqlite32.dll na pasta System32 do Windows, e a forma de instanciar é um pouco diferentes dos outros adaptadores, neste caso é apenas necessário passar o caminho do arquivo do banco para ser aberto (ou criado), como mostra a **Listagem 2**.

Listagem 2. Instanciando o Adaptador SQLite

```
MinhaConexao := TSQLiteNativeConnectionAdapter.Create('C:\DataBase\SQLite\
MeuBanco.sdb');
```

É permitido o uso de um banco de dados já existente, ou é possível deixá-lo criar o banco. Baseando-se pela estrutura de mapeamento criada. Para optar pela segunda opção, basta incluir a Unit Aurelius.Engine.DatabaseManager, e fazer uso da classe TDataBaseManager, como mostra a **Listagem 3**. Essa classe possui métodos que tratam da estrutura de um banco de dados, por exemplo, o método BuildDatabase, que irá criar a estrutura do referido banco de dados representado por um objeto de conexão.

Listagem 3. Criando o Banco de Dados com o Aurelius

```
Var
  DBManager:TDataBaseManager;
Begin
  DBManager:= TDataBaseManager.Create(MinhaConexao);
  DBManager.BuildDatabase;
  DBManager.Free;
End;
```

Existe também a opção Validate, esta, permite que o Aurelius execute uma comparação entre a atual estrutura das classes e a do banco de dados, e se houver alguma mudança ele modifica a estrutura do banco. Esse comportamento reflete uma boa prática no desenvolvimento OO. Se ainda não existe um banco de dados, não comece sua aplicação OO por ele. Comece pela representação do negócio, em classes. Dessa forma toda a análise não estará limitada a apenas dados. A **Listagem 4** mostra como fazer uso do método Validate.

Nas linhas 2 e 3 adicionamos as units necessárias. Nas linhas 5 e 6 temos a declaração das variáveis DBManager que é do tipo TDatabaseManager. Já SchemaMessage é do tipo TSchemaMessage. Esse tipo representa uma mensagem que é gerada no momento da atualização da estrutura do banco de dados.

Quando o método ValidateDatabase é executado (linha 9), ele retorna true, se tudo está correto e false quando algum problema é levantado. Caso tenha acontecido algum problema, a lista dos problemas é gerada nas propriedades Warnings, Errors e Actions, de acordo com seu tipo, assim, é possível percorrer essas listas (linhas 14, 16 e 18) exibindo o que aconteceu de errado.

O Aurelius também permite que o banco de dados seja apagado através do método DestroyDatabase (**Listagem 5**), mas é importante ressaltar que as tabelas que serão apagadas do banco de dados são aquelas que foram mapeadas e criadas através do Aurelius.

Dialeto SQL	Identificador	Unit	Web Site
Absolute Database	AbsoluteDB	Aurelius.Sql.Absolut eDB	http://www.componentace.com
DB2	DB2	Aurelius.Sql.DB2	http://www.ibm.com
ElevateDB	ElevateDB	Aurelius.Sql.ElevateDB	http://www.elevatesoftware.com
Firebird	Firebird	Aurelius.Sql.Firebird	http://www.firebirdsql.org
Interbase	Interbase	Aurelius.Sql.Interbase	http://www.embarcadero.com
Microsoft SQL Server	MSSQL	Aurelius.Sql.MSSQL	http://www.microsoft.com/sqlserver
MySQL	MySQL	Aurelius.Sql.MySQL	http://www.mysql.com
NexusDB	NexusDB	Aurelius.Sql.NexusDB	http://www.nexusdb.com
Oracle	Oracle	Aurelius.Sql.Oracle	http://www.oracle.com
PostgreSQL	PostgreSQL	Aurelius.Sql.PostgreSQL	http://www.postgresql.org
SQLite	SQLite	Aurelius.Sql.SQLitxce	http://www.sqlite.org

Tabela 2. Dialetos suportados pelo Aurelius

Criando o exemplo

Para este artigo iremos utilizar o banco de dados Firebird, portanto faremos o uso do adaptador do dbExpress que pode ser encontrado na Unit Aurelius.Drivers.dbExpress e também utilizaremos o dialeto SQL do Firebird, então faremos o uso da Unit Aurelius.Sql.Firebird em nosso projeto.

Crie um novo banco de dados, mas deixe-o vazio, inicie um novo projeto no Delphi, adicione uma nova Unit em seu projeto, a renomeie como ServiceU. Nela vamos configurar a nossa conexão com o banco de dados. Na **Listagem 6**

pode-se conferir o código para criar a SQLConnection, dessa forma poderemos utilizá-la para instanciar as classes do domínio e efetuar operações no banco de dados.

Listagem 4. Utilizando Validation em um Banco de Dados já existente

```
01 uses
02 Aurelius.Engine.DatabaseManager,
03 Aurelius.Schema.Messages;
04 var
05  DBManager: TDatabaseManager;
06  SchemaMessage: TSchemaMessage;
07 begin
08  DBManager := TDatabaseManager.Create(MyConnection);
09  if DBManager.ValidateDatabase then
10   WriteLn('Database strucuture is valid.')
11  else
12  begin
13   WriteLn(Format('Invalid database structure. %d Errors, %d Warnings,
      % d Actions',
     [DBManager.ErrorCount, DBManager.WarningCount,
     DBManager.ActionCount]));
14   for SchemaMessage in DBManager.Warnings do
15      WriteLn('Warning: ' + SchemaMessage.Text);
16   for SchemaMessage in DBManager.Errors do
17    WriteLn('Warning: ' + SchemaMessage.Text);
18   for SchemaMessage in DBManager.Actions do
19    WriteLn('Warning: ' + SchemaMessage.Text);
20  end;
21  DBManager.Free;
22 end;
```

Listagem 5. Apagando um banco de dados atravéns do Aurelius

```
Var
  DBManager:TDataBaseManager;
Begin
  DBManager:= TDataBaseManager.Create(MinhaConexao);
  DBManager.DestroyDatabase;
  DBManager.Free;
End;
```

Listagem 6. Função para cria a SQLConnection

```
01 begin
02  if not Assigned(FSQLConnection) then
03  begin
04   FSQLConnection := TSQLConnection.Create(nil);
05   with FSQLConnection do
06   begin
07    Params.LoadFromFile('Config.ini');
08    ConnectionName  := Params.Values['Driver'];
09    DriverName     := Params.Values['Driver'];
10    LibraryName    := Params.Values['LibraryName'];
11    GetDriverFunc  := Params.Values['GetDriverFunc'];
12    VendorLib      := Params.Values['VendorLib'];
13    LoginPrompt    := False;
14   end;
15  end;
16  Result := FSQLConnection;
17  try
18   try
19    Result.Connected := True;
20   except
21    raise Exception.Create('Erro ao conectar-se ao Banco de Dados');
22   end;
23  finally
24   Result.Connected := False;
25  end;
```

Note que a configuração é bem semelhante com a forma convencional de se conectar com um banco de dados, basta instanciar o TSQLConnection (linha 4) e configurar os parâmetros de conexão (linhas 5 a 14), o próximo passo é fazer com que o Aurelius monte as tabelas no banco de dados conforme estão estruturadas nas classes mapeadas, a **Listagem 7** mostra esse código.

Listagem 7. Função para registrar as classes mapeadas e validar o banco de dados

```
01 var
02  MapSetup: TMappingSetup;
03 begin
04  if not Assigned(FInstance) then
05  begin
06   FInstance := TDBExpressConnectionAdapter.Create (GetSQLConnection,
     'Firebird', False);
07
08   MapSetup := TMappingSetup.Create;
09   MapSetup.MappedClasses.RegisterClass(TVenda);
10   MapSetup.MappedClasses.RegisterClass(TProduto);
11   MapSetup.MappedClasses.RegisterClass(TItemVenda);
12
13   TMappingExplorer.
14     ReplaceDefaultInstance(TMappingExplorer.Create(MapSetup));
15   FreeAndNil(MapSetup);
16
17   with TDatabaseManager.Create(Finstance) do
18   begin
19    if not ValidateDatabase then
20     UpdateDatabase;
21    Free;
22   end;
23  end;
24  Result := FInstance;
```

Note a variável MapSetup representa o mecanismo do Aurelius responsável pelo registro dos mapeamentos. Logo após ser instanciada (linha 8), as classes que estão mapeadas são registras nela. Na linha 13 é "informado" para o Aurelius que as classes que estão registradas serão parte da estrutura no banco que será criado ou atualizado.

Nota

Este recurso de registrar classes para criar a estrutura do banco de dados foi integrado ao Aurelius apenas na versão 2.0, nas versões anteriores esse procedimento é feito de outra forma.

Para construir a classe de conexão, devemos utilizar as seguintes Units do Aurelius:

- Aurelius.Drivers.Interfaces;
- Aurelius.Drivers.DbExpress;
- Aurelius.Sql.Firebird;
- Aurelius.Schema.Firebird;
- Aurelius.Mapping.Setup;
- Aurelius.Mapping.Explorer;
- Aurelius.Engine.ObjectManager;
- Aurelius.Engine.DataBaseManager.

Para não ficar muito extenso você pode conferir o código completo da classe de conexão no exemplo disponível para download.

A primeira etapa está pronta, agora devemos criar as classes e fazer o mapeamento para que se torne possível utilizarmos o Aurelius, mas antes uma breve explicação sobre mapeamento por Automapping e Custom Attributes, ID e Sequence.

Automapping e Custom Attributes

Podemos fazer o mapeamento com base em um banco de dados já existente e também é possível mapear as classes e criar o banco com base nesse mapeamento.

O recurso de Automapping realiza o mapeamento por convenção, ou seja, o Aurelius vai atribuir o nome de sua classe como o nome da tabela, e o nome de cada atributo para cada campo, por exemplo, se a classe se chama "TCliente", o nome da tabela será "Cliente", e para um atributo, por exemplo "FNome", o Aurelius vai atribuir "Nome" ao título do campo da tabela do banco de dados.

Já no mapeando através de Custom Attributes devemos indicar o nome da tabela equivalente à aquela classe com a tag "[Table(NOME_TABELA')]", já para os atributos devemos também indicar qual é o campo equivalente com a tag "[Column('NOME_CAMPO')]", dessa forma o Aurelius entenderá como ele deve montar a estrutura do banco de dados.

Para ambos os modos de fazer o mapeamento, é obrigatório definir que a classe é uma entidade, para que ela possa se "transformar" em uma tabela do banco de dados. Isso deve ser feito através da tag [Entity], mais adiante iremos entender como montar exatamente essa estrutura em uma classe. A Unit do Aurelius que deve ser utilizada para o mapeamento de classes é a Aurelius.Mapping.Attributes.

ID e Sequence

O ID é o identificador do objeto, é extremamente importante definir um atributo ID para sua classe, pois no momento de criar o banco de dados é essa definição que vai indicar a chave primária da tabela. Para declarar o ID da classe basta adicionar a Tag "[Id('FId', TIdGenerator.IdentityOrSequence)]", onde "FID" é o nome do atributo no qual será definido como ID da classe. A geração dos identificadores é representada através do enumerador TIdGenerator. Como estamos utilizando um banco de dados Firebird, utilizamos TIdGenerator.IdentityOrSequence, que dá suporte ao uso de generators – ou sequences – nativamente suportados. Então é preciso definir o nome do gerenator a ser utilizado, isso é feito utilizando a Tag "[Sequence('NOME_GENERATOR')]".

Com esses conceitos já é possível iniciar um exemplo prático mapeando classes com o Aurelius, é sempre uma boa prática quando se desenvolve uma aplicação Orientada a Objetos desenvolver o diagrama de classes, para ter uma ideia geral do que será necessário desenvolver e ter um sistema bem estruturado, portanto em nossa aplicação prática seguiremos o seguinte diagrama da **Figura 1**.

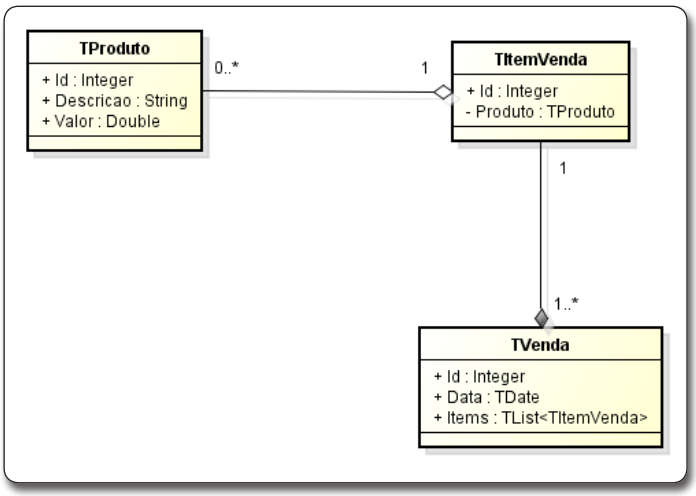


Figura 1. Diagrama de classes

Iniciaremos construção das classes pela classe TProduto, note que nela apenas existem três atributos, portanto podemos fazer o mapeamento através do modo mais simples, o Automapping. A seguir, na **Listagem 8**, o código da classe TProduto mapeada através do Automapping.

Listagem 8. Classe TProduto mapeada com Automapping

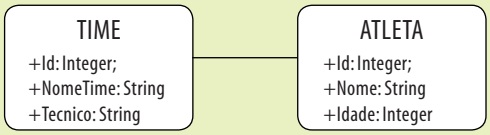
```
01 [Entity]
02 [Automapping]
03 [Sequence('GEN_PRODUTO')]
04 [Id('FId',TIdGenerator.IdentityOrSequence)]
05 TProduto = class
06 private
07   [Column('ID',[TColumnProp.Unique,TColumnProp.Required,
     TColumnProp.NoUpdate])]
08   FId: Integer;
09   FDescricao: String;
10   FValor: Double;
11   procedure SetDescricao(const Value: String);
12   procedure SetId(const Value: Integer);
13   procedure SetValor(const Value: Double);
14 public
15   property Id: Integer read FId write SetId;
16   property Descricao: String read FDescricao write SetDescricao;
17   property Valor: Double read FValor write SetValor;
18 end;
```

Note que definimos o Sequence pertencente a essa classe (linha 3), neste caso é o 'GEN_PRODUTO'. Definimos também o identificador (linha 7) da classe na Tag logo a cima do atributo FId, esse atributo define que este será um campo único, obrigatório e que não sofrerá modificações com futuras mudanças, afinal, este campo representará a chave primária no banco de dados.

Agora iremos criar a classe TItemVenda, nesta classe temos uma Agregação (**Nota do DevMan 2**) a classe TProduto, portanto vamos fazer esse mapeamento utilizando a técnica Custom Attributes, vamos conferir a classe TItemVenda mapeada com Custom Attributes e com essa agregação na **Listagem 9**.

Nota do DevMan 2

Agregação: Podemos definir uma agregação quando em uma relação Parte-Todo perdermos o Todo e ainda faz sentido manter a parte no banco de dados, vejamos um exemplo na relação entre as tabelas entre Time e Atleta:



Um time é formado por atletas, ou seja, os atletas são parte integrante de um time, mas os atletas existem independentemente de um time existir. Nesse caso, chamamos esse relacionamento de AGREGAÇÃO.

Note que logo acima do atributo que representa a composição (FProduto: TProduto) (**Nota do DevMan 3**) temos a definição de algumas Tags.

A primeira Tag, na linha 9, diz que este atributo será uma chave estrangeira no banco de dados, o primeiro parâmetro dessa tag define os atributos da associação, neste caso uma associação obrigatória, o segundo parâmetro define como serão propagadas as alterações para os objetos propagados.

Listagem 9. Classe TItemVenda mapeada com Custom Attribute

```
01 [Entity]
02 [Table('ITEM_VENDA')]
03 [Sequence('GEN_ITEM_VENDA')]
04 [Id('FId', TIdGenerator.IdentityOrSequence)]
05 TItemVenda = class
06 private
07   [Column('ITEM_ID', [TColumnProp.Unique, TColumnProp.Required,
     TColumnProp.NoUpdate])]
08   FId: Integer;
09   [Association([TAssociationProp.Required], [TCascadeType.SaveUpdate])]
10   [JoinColumn('ITEM_PRODUTO_ID', [TColumnProp.Required])]
11   FProduto: TProduto;
12   procedure SetId(const Value: Integer);
13   procedure SetProduto(const Value: TProduto);
14 public
15   property Id: Integer read FId write SetId;
16   property Produto: TProduto read FProduto write SetProduto;
17 end;
```

A segunda, na linha 10, informa o nome do campo que ele terá no banco de dados, além de dizer que é um campo obrigatório.

Vamos agora mapear a classe TVenda, observe que essa classe é uma composição com a classe TItenVenda, e temos um atributo Itens, que é uma lista de TItenVenda, vejamos agora como faremos o mapeamento dessa classe (**Listagem 10**).

CURSOS ONLINE

A Revista Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Clube Delphi

CONHEÇA ALGUNS DOS CURSOS:

- **Curso de Multicamadas com Delphi e DataSnap**
- **Delphi para Iniciantes**
- **Criando componente Boleto em Delphi**
- **Loja Virtual em Delphi Prism**

Para mais informações :

www.devmedia.com.br/cursos/delphi

(21) 3382-5038

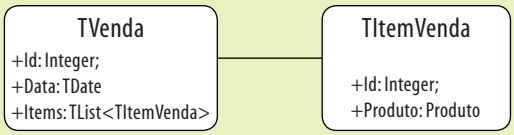
DevMedia

Listagem 10. Classe TVenda mapeada com Custom Attribute

```
01 [Entity]
02 [Table('VENDA')]
03 [Sequence('GEN_VENDA')]
04 [Id('FId', TIdGenerator.IdentityOrSequence)]
05 TVenda = class
06 private
07   [Column('VENDA_ID', [TColumnProp.Unique, TColumnProp.Required,
     TColumnProp.NoUpdate])]
08   FId: Integer;
09   [Column('VENDA_DATA', [TColumnProp.Required])]
10   FData: TDate;
11   [ManyValuedAssociation([], CascadeTypeAll)]
12   [ForeignJoinColumn('ITEM_VENDA_ID', [TColumnProp.Required])]
13   FItems: TList<TItemVenda>;
14   procedure SetData(const Value: TDate);
15   procedure SetId(const Value: Integer);
16   procedure SetItems(const Value: TList<TItemVenda>);
17 public
18   property Id: Integer read FId write SetId;
19   property Data: TDate read FData write SetData;
20   property Items: TList<TItemVenda> read FItems write SetItems;
21 end;
```

Nota do DevMan 3

A Composição é uma relação mais forte, ou seja, neste caso em uma relação Parte-Todo NÃO faz sentido ter a parte quando se perde o todo, vejamos o exemplo entre a relação Pedido e Item Pedido:



Assim, uma venda é composta por um ou vários itens, mas um item NÃO é item de uma venda se não existir essa venda. Assim, chamamos esse relacionamento de COMPOSIÇÃO.

Veja que na linha 11 temos o atributo ManyValuedAssociation que indica que este campo é uma coleção de valores do tipo TItenVenda e o atributo ForeignJoinColumn da linha 12 mostra que ela será uma chave estrangeira na tabela ItenVenda.

Conclusão

Neste artigo abordamos o mapeamento objeto relacional com Aurelius. Vimos como uma agregação ou composição pode ser mapeada, com isso, o próprio framework se responsabiliza pela geração correta das sentenças SQL para já carregar automaticamente o "mestre-detalhe" criado no lado relacional. Essa é uma das vantagens de um ORM, a ocultação, encapsulamento das dificuldades do mundo relacional. A abordagem de desenvolvimento orientado a objetos demanda uma maior análise para criação das classes de negócio, que em uma boa modelagem são mais do que simplesmente espelhos de tabelas. É importante que o desenvolvedor Delphi tenha isso em mente. Quando se decide desenvolver um sistema orientado a objetos, seus objetos são mais do que tabelas no banco de dados, eles devem conter comportamento, devem fazer parte de um todo que represente um comportamento.

Murilo Aparecido Panosso
murilopanosso@gmail.com
Cursando Tecnologia em Analise e Desenvolvimento de Sistemas pela Fatec de Taquaritinga, atualmente compõe a equipe de programadores na empresa SiplanControl-M (www.siplancontrolm.com.br).

Site do Aurelius
http://www.tmssoftware.com/site/aurelius.asp

Manual do Aurelius em PDF
http://www.tmssoftware.com.br/aurelius/doc/aurelius_manual.pdf

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

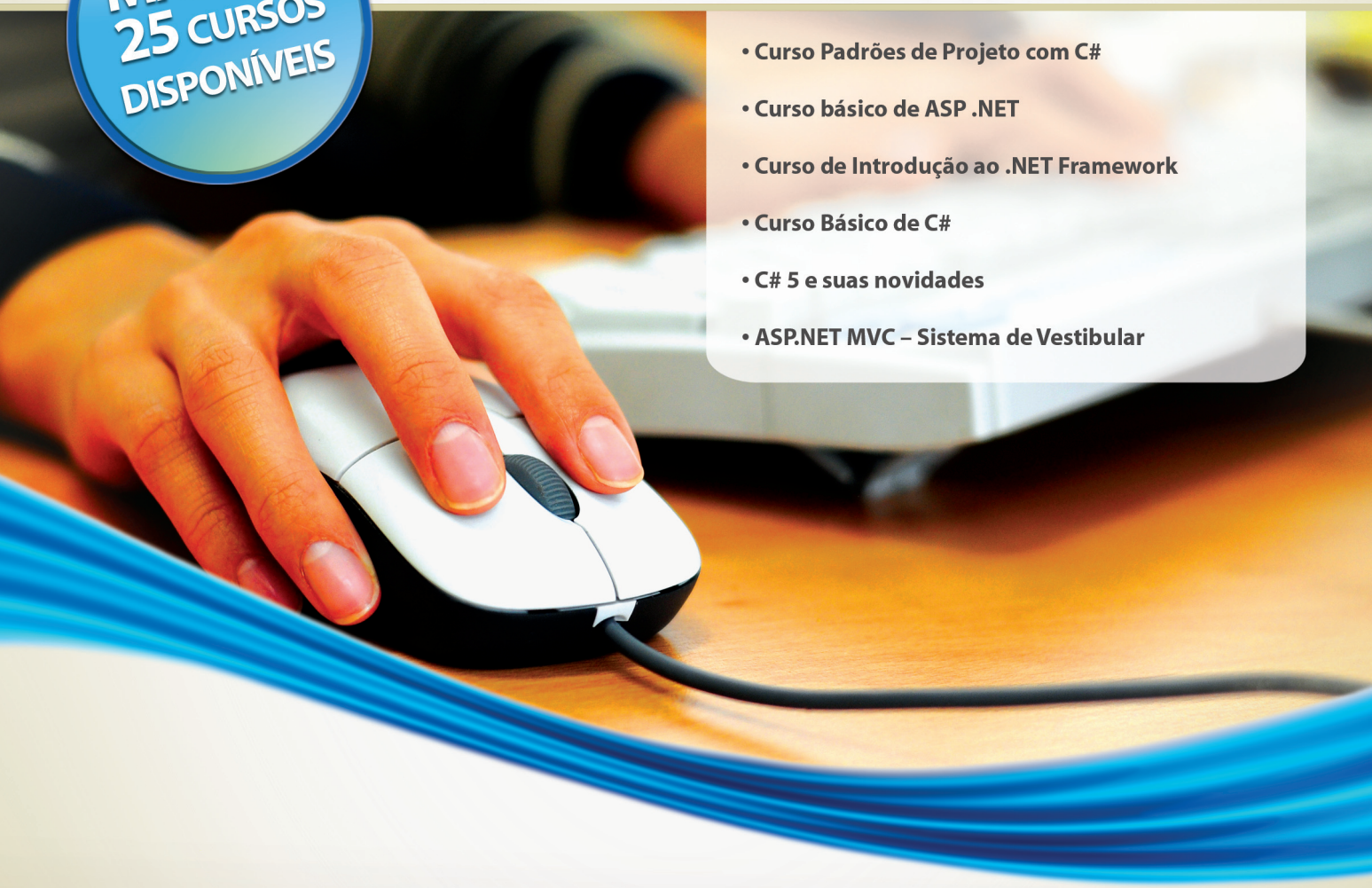
CURSOS ONLINE

.net magazine

A Revista .net Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

MAIS de 25 CURSOS DISPONÍVEIS

CONHEÇA OS CURSOS MAIS RECENTES:



- Curso Padrões de Projeto com C#
- Curso básico de ASP .NET
- Curso de Introdução ao .NET Framework
- Curso Básico de C#
- C# 5 e suas novidades
- ASP.NET MVC – Sistema de Vestibular

Para mais informações :

www.devmedia.com.br/curso/netmagazine

(21) 3382-5038

http://www.devmedia.com.br

DevMedia

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486